pyspatial Documentation

Release 0.2.1

Aman Thakral

February 01, 2017

Contents

Contents:

CHAPTER 1

Overview

pyspatial is python package to provide data structures on top of gdal/ogr. Its core use cases have been around simplifying geospatial data science workflows in Python. The 3 core data stuctures are:

- VectorLayer: a collection of geometries with pandas like manipulation. Each geometry is an osgeo.ogr.Geometry object. For an object reference see this link.
- RasterDataset: an abstraction of a spatial raster (both tiled on untiled) to support querying of pixels that intersect with shapes.
- TiledWebRaster: an abstraction of a tiled spatial raster typically used for visualization on the web (e.g. openlayers or google maps) (Still in testing)
- RasterBand: a numpy array representation of a raster with spatial metadata (in memory only, no tiled support).

pyspatial makes it easy to read, analyze, query, and visualize spatial data in both vector and raster form. It brings the familiarity of pandas to working with vector data, and provides querying capability similar to PostGIS for both vector and raster data. Since it uses GDAL for much of the computations, the performance is quite good. Based on the authors' experience, the performance has been significantly better than PostGIS, and orders of magnitude faster than similar libraries in R.

Documentation is available here

1.1 Library Highlights

- Battle tested: we use it for our day-to-day work, and for processing all the data behind AcreValue. In fact, all of our PostGIS workflows have been migrated to pyspatial.
- Read/write both raster and vector data (including support for http/s3/google cloud sources). Also convert to/from shapely/gdal/ogr/numpy objects seamlessly.
- Fast spatial queries since it leverages GDAL and libspatialindex/RTree. For extracting vector data from a raster, the library is 60x - 100x faster than R.
- Integration of vector/raster data structures to make interoperation seamless.
- Pandas-like API for working with collections of geometries.
- First class support for spatial projections. The data structures are spatial projection aware, and allow you to easily transform between projections.
- When performing operations between data sources, the data will automatically be reprojected intelligently. No more spatial projection management!
- Integrated interactive visualization within IPython (via Leaflet). Plots markers, geometries, and choropleths too!

1.2 Examples

Plese see this link.

1.3 Questions?

Send us a message on google groups: pyspatial-users@googlegroups.com

1.4 Development

- Python code: pip install -e /path/to/pyspatial
- Cython code: python setup.py build_ext –inplace
- Tests: nosetests -v

1.5 TODOs

- Adjust timed tests. Currently calibrated to a early Macbook Pro 15" with Core i7. These tend to fail on many other machines. Should either remove the @timed decorators, or figure out what a reasonable time is for the tests.

1.6 Known Issues

- In ogr, when you get the centroid of a geometry (e.g. cent = my_geometry.Centroid()), cent does not inherit the spatial reference. It needs to be reassigned using cent.AssignSpatialReference(my_geometry.GetSpatialReference())
- VectorLayer object does not support a Float64Index
- If you encouter:
  - "TypeError: object of type 'Geometry' has no len()", most likely you have duplicate values in your index. Make sure your index is unique.
- On some environments, calling shape.Intersection(point) for certain shape/point combinations causes python to crash. See this gist

1.7 Contributors

- Aman Thakral (Lead, github: aman-thakral)
- Josh Daniel (github: joshdaniel)
- Chris Seifert (github: caseifert)
- Ron Potok (github: rpotok)
- Sandra Guteg (github: guetgs)

- Emma Fuller (github: emfuller)
- Alan Glennon (github: glennon)
- James Russell (github: jamesdrussell)

CHAPTER 2

LICENSE

CHAPTER 3

INSTALLATION

3.1 OSX

- Scripts provided are based off of homebrew (copy and paste into terminal). Should not install if you already have MacPorts.
- No support for MacPorts is provided.

```
# Install Homebrew
ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"

brew install geos
brew install spatialindex

# Install latest GDAL (1.11.2)
brew install gdal

## Alternatively, install GDAL 2.0 (Still in testing)
# brew install
# https://gist.githubusercontent.com/brianreavis/261cf46b44669366df9c/raw/aa7f2f2a8a511975d7d1dae9e5
```

3.2 Debian/Ubuntu

```
# install the system dependencies
sudo add-apt-repository -y ppa:ubuntugis/ppa
sudo apt-get update
sudo apt-get install -y libgdal-dev
sudo apt-get install -y libspatialindex-dev
sudo apt-get install -y libblas-dev \
liblapack-dev libatlas-base-dev gfortran libfreetype6-dev

# Optional: create python virtual environment
virtualenv venv
source venv/bin/activate

# update GDAL version in requirements.txt to match system version
# (GDAL 2.0.2 doesn't seem to be available in any PPA)
# GDAL==1.11.2

# Configure GDAL before installing
```

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

# install python dependencies
pip install numpy scipy
pip install -r requirements-dev.txt
pip install -r requirements.txt
pip install -e /path/to/pyspatial

# run the tests
nosetests -v
```

3.3 Python

3.3.1 PyPI

```
pip install pyspatial
```

3.3.2 Latest

```
git clone https://github.com/granularag/pyspatial.git
cd pyspatial
pip install -r requirements.txt
pip install .
```

3.4 Appendix

3.4.1 Compiling GDAL from source

- A good overview is provided here: https://docs.djangoproject.com/en/1.9/ref/contrib/gis/install/geolibs/
- More detailed information can be found here: https://trac.osgeo.org/gdal/wiki/BuildHints

1. If you don't have root access, you should download the source and build packages like

- $ ./configure –prefix ~/local
- To make the binaries available add the following to your bashrc

```
export HOME_DIR=/my/home/dir
export PATH=$PATH:$HOME_DIR/local/bin
```

- To build gdal (assume geos installed in /usr/local), in non-standard localtion: * $ export HOME_DIR=/my/home/dir * $ ./configure –enable-64bit –prefix $HOME_DIR/local –with-includes=$HOME_DIR/local/include/ –with-libs=$HOME_DIR/local/lib/ –with-sqlite=no –with-geos=/usr/local/bin/geos-config –with-opengl=no –with-cairo=no –with-freetype=no –with-lapack –with-blas –with-readline
- In your scripts/bashrc:

```
export HOME_DIR=/my/home/dir
export GDALHOME=$HOME_DIR/local/
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$USER_HOME/local/lib/
```

CHAPTER 4

pyspatial

4.1 pyspatial package

4.1.1 Submodules

4.1.2 pyspatial.dataset module

pyspatial.dataset.**dumps**(*x*, *double_precision=6*)

pyspatial.dataset.**get_sample_pt**(*s*)

pyspatial.dataset.**get_type**(*s*, *type_map={<class 'pandas.tslib.Timestamp'>: 'datetime', <type 'datetime.datetime'>: 'datetime', <type 'datetime.date'>: 'datetime', <type 'float'>: 'number', <type 'basestring'>: 'text', <type 'int'>: 'number', <type 'bool'>: 'bool'}*)

pyspatial.dataset.**to_dict**(*df*, *hidden=None*, *not_visible=None*, *labels=None*, *types=None*)

Convert a DataFrame into a Dataset dictionary. This can later be serialized to json using the 'dumps' method found in this module. Types are either inferred or taken from the 'type' parameter. If the types cannot be identified, those columns will not be included. Note, if all the values of a column are NaN, the column will also be removed.

Parameters

- **hidden** (*list (default=None)*) – Columns in df to mark as hidden
- **not_visible** (*list (default=None)*) – Columns in df to not mark as visible
- **labels** (*dict*) – Keys are the column names and values are the 'label' attribute
- **types** (*dict*) – Keys are the column names and values are the 'type' attribute

Returns

Return type Python dictionary with the attributes 'data', 'schema', 'index'

4.1.3 pyspatial.fileutils module

class pyspatial.fileutils.**GSOpenRead**(*read_key*)

Bases: object

Implement streamed reader from GS.

read(*size=None*)
Read a specified number of bytes from the key.

class pyspatial.fileutils.**GSOpenWrite**(*outkey*)
Bases: object

Context manager for writing into GS files.

close()

write(*b*)
Write the given bytes (binary string) into the GS file from constructor.

pyspatial.fileutils.**get_path**(*path*)

pyspatial.fileutils.**open**(*path*, *mode='rb'*, ***kw*)

pyspatial.fileutils.**parse_uri**(*uri*)

4.1.4 pyspatial.globalmaptiles module

globalmaptiles.py

Global Map Tiles as defined in Tile Map Service (TMS) Profiles

Functions necessary for generation of global tiles used on the web. It contains classes implementing coordinate conversions for:

- **GlobalMercator (based on EPSG:900913 = EPSG:3785)** for Google Maps, Yahoo Maps, Microsoft Maps compatible tiles
- **GlobalGeodetic (based on EPSG:4326)** for OpenLayers Base Map and Google Earth compatible tiles

More info at:

http://wiki.osgeo.org/wiki/Tile_Map_Service_Specification http://wiki.osgeo.org/wiki/WMS_Tiling_Client_Recommendation
http://msdn.microsoft.com/en-us/library/bb259689.aspx http://code.google.com/apis/maps/documentation/overlays.html#Google_Maps

Created by Klokan Petr Pridal on 2008-07-03. Google Summer of Code 2008, project GDAL2Tiles for OSGEO.

In case you use this class in your product, translate it to another language or find it usefull for your project please let me know. My email: klokan at klokan dot cz. I would like to know where it was used.

Class is available under the open-source GDAL license (www.gdal.org).

class pyspatial.globalmaptiles.**GlobalGeodetic**(*tileSize=256*)
Bases: object

Functions necessary for generation of global tiles in Plate Carre projection, EPSG:4326, “unprojected profile”.

Such tiles are compatible with Google Earth (as any other EPSG:4326 rasters) and you can overlay the tiles on top of OpenLayers base map.

Pixel and tile coordinates are in TMS notation (origin [0,0] in bottom-left).

What coordinate conversions do we need for TMS Global Geodetic tiles?

> Global Geodetic tiles are using geodetic coordinates (latitude,longitude) directly as planar coordinates XY (it is also called Unprojected or Plate Carre). We need only scaling to pixel pyramid and cutting to tiles. Pyramid has on top level two tiles, so it is not square but rectangle. Area [-180,-90,180,90] is scaled to 512x256 pixels. TMS has coordinate origin

(for pixels and tiles) in bottom-left corner. Rasters are in EPSG:4326 and therefore are compatible with Google Earth.

LatLon <-> Pixels <-> Tiles

WGS84 coordinates Pixels in pyramid Tiles in pyramid

lat/lon XY pixels Z zoom XYZ from TMS

EPSG:4326 .—-. —-

/ <-> /——/ <-> TMS / /————/

—- /———————/

WMS, KML Web Clients, Google Earth TileMapService

LatLonToPixels(*lat*, *lon*, *zoom*)
Converts lat/lon to pixel coordinates in given zoom of the EPSG:4326 pyramid

PixelsToTile(*px*, *py*)
Returns coordinates of the tile covering region in pixel coordinates

Resolution(*arc/pixel) for given zoom level (measured at Equator*)

TileBounds(*tx*, *ty*, *zoom*)
Returns bounds of the given tile

class pyspatial.globalmaptiles.**GlobalMercator**(*tileSize=256*)
Bases: object

Functions necessary for generation of tiles in Spherical Mercator projection, EPSG:900913 (EPSG:gOOglE, Google Maps Global Mercator), EPSG:3785, OSGEO:41001.

Such tiles are compatible with Google Maps, Microsoft Virtual Earth, Yahoo Maps, UK Ordnance Survey OpenSpace API, ... and you can overlay them on top of base maps of those web mapping applications.

Pixel and tile coordinates are in TMS notation (origin [0,0] in bottom-left).

What coordinate conversions do we need for TMS Global Mercator tiles:

```
     LatLon      <->       Meters      <->     Pixels    <->       Tile

 WGS84 coordinates   Spherical Mercator  Pixels in pyramid  Tiles in pyramid
     lat/lon            XY in metres     XY pixels Z zoom      XYZ from TMS
    EPSG:4326           EPSG:900913
     .----.              ---------               --                TMS
    /      \     <->     |       |     <->     /----/    <->      Google
    \      /              |       |           /--------/          QuadTree
     -----               ---------         /------------/
   KML, public         WebMapService         Web Clients      TileMapService
```

What is the coordinate extent of Earth in EPSG:900913?

[-20037508.342789244, -20037508.342789244, 20037508.342789244, 20037508.342789244] Constant 20037508.342789244 comes from the circumference of the Earth in meters, which is 40 thousand kilometers, the coordinate origin is in the middle of extent. In fact you can calculate the constant as: 2 * math.pi * 6378137 / 2.0 $ echo 180 85 | gdaltransform -s_srs EPSG:4326 -t_srs EPSG:900913 Polar areas with abs(latitude) bigger then 85.05112878 are clipped off.

What are zoom level constants (pixels/meter) for pyramid with EPSG:900913?

whole region is on top of pyramid (zoom=0) covered by 256x256 pixels tile, every lower zoom level resolution is always divided by two initialResolution = 20037508.342789244 * 2 / 256 = 156543.03392804062

What is the difference between TMS and Google Maps/QuadTree tile name convention?

The tile raster itself is the same (equal extent, projection, pixel size), there is just different identification of the same raster tile. Tiles in TMS are counted from [0,0] in the bottom-left corner, id is XYZ. Google placed the origin [0,0] to the top-left corner, reference is XYZ. Microsoft is referencing tiles by a QuadTree name, defined on the website: http://msdn2.microsoft.com/en-us/library/bb259689.aspx

The lat/lon coordinates are using WGS84 datum, yeh?

Yes, all lat/lon we are mentioning should use WGS84 Geodetic Datum. Well, the web clients like Google Maps are projecting those coordinates by Spherical Mercator, so in fact lat/lon coordinates on sphere are treated as if the were on the WGS84 ellipsoid.

From MSDN documentation: To simplify the calculations, we use the spherical form of projection, not the ellipsoidal form. Since the projection is used only for map display, and not for displaying numeric coordinates, we don't need the extra precision of an ellipsoidal projection. The spherical projection causes approximately 0.33 percent scale distortion in the Y direction, which is not visually noticable.

How do I create a raster in EPSG:900913 and convert coordinates with PROJ.4?

You can use standard GIS tools like gdalwarp, cs2cs or gdaltransform. All of the tools supports -t_srs 'epsg:900913'.

For other GIS programs check the exact definition of the projection: More info at http://spatialreference.org/ref/user/google-projection/ The same projection is degined as EPSG:3785. WKT definition is in the official EPSG database.

Proj4 Text: +proj=merc +a=6378137 +b=6378137 +lat_ts=0.0 +lon_0=0.0 +x_0=0.0 +y_0=0 +k=1.0 +units=m +nadgrids=@null +no_defs

Human readable WKT format of EPGS:900913:

PROJCS["Google Maps Global Mercator",

GEOGCS["WGS 84",

DATUM["WGS_1984",

SPHEROID["WGS 84",6378137,298.2572235630016,
AUTHORITY["EPSG","7030"]],

AUTHORITY["EPSG","6326"]],

PRIMEM["Greenwich",0], UNIT["degree",0.0174532925199433], AUTHORITY["EPSG","4326"]],

PROJECTION["Mercator_1SP"], PARAMETER["central_meridian",0], PARAMETER["scale_factor",1], PARAMETER["false_easting",0], PARAMETER["false_northing",0], UNIT["metre",1,

AUTHORITY["EPSG","9001"]]]

GoogleTile(*tx*, *ty*, *zoom*)
Converts TMS tile coordinates to Google Tile coordinates

LatLonToMeters(*lat*, *lon*)
Converts given lat/lon in WGS84 Datum to XY in Spherical Mercator EPSG:900913

MetersToLatLon (*mx*, *my*)
Converts XY point from Spherical Mercator EPSG:900913 to lat/lon in WGS84 Datum

MetersToPixels (*mx*, *my*, *zoom*)
Converts EPSG:900913 to pyramid pixel coordinates in given zoom level

MetersToTile (*mx*, *my*, *zoom*)
Returns tile for given mercator coordinates

PixelsToMeters (*px*, *py*, *zoom*)
Converts pixel coordinates in given zoom level of pyramid to EPSG:900913

PixelsToRaster (*px*, *py*, *zoom*)
Move the origin of pixel coordinates to top-left corner

PixelsToTile (*px*, *py*)
Returns a tile covering region in given pixel coordinates

QuadTree (*tx*, *ty*, *zoom*)
Converts TMS tile coordinates to Microsoft QuadTree

Resolution (*meters/pixel) for given zoom level (measured at Equator*)

TileBounds (*tx*, *ty*, *zoom*)
Returns bounds of the given tile in EPSG:900913 coordinates

TileLatLonBounds (*tx*, *ty*, *zoom*)
Returns bounds of the given tile in latutude/longitude using WGS84 datum

ZoomForPixelSize (*pixelSize*)
Maximal scaledown zoom of the pyramid closest to the pixelSize.

4.1.5 pyspatial.io module

exception pyspatial.io.**PyspatialIOError**
Bases: exceptions.Exception

pyspatial.io.**create_zip** (*path*)

pyspatial.io.**get_gdal_datasource** (*path*)

pyspatial.io.**get_ogr_datasource** (*path*, *use_streaming=False*)

pyspatial.io.**get_path** (*path*, *use_streaming=False*)
Read a shapefile from local or an http source

pyspatial.io.**get_schema** (*df*)

pyspatial.io.**read_in_chunks** (*file_object*, *chunk_size=4096*)
Lazy function (generator) to read a file piece by piece. Default chunk size: 4k.

pyspatial.io.**upload** (*local_filename*, *remote_path*, *remove_local=False*)
Upload a local file to a remote location. Currently, only s3/gs is suppported

pyspatial.io.**uri_to_string** (*uri*)

pyspatial.io.**write_shapefile** (*vl*, *path*, *name=None*, *df=None*, *driver='ESRI Shapefile'*)

pyspatial.io.**zipdir** (*path*, *ziph*)

4.1.6 pyspatial.py3 module

4.1.7 pyspatial.raster module

class pyspatial.raster.**RasterBand**(*ds*, *band_number=1*)
Bases: *pyspatial.raster.RasterBase*, numpy.ndarray

save(*path*, *format='GTiff'*)

save_png(*path*)

to_gdal(*driver='MEM'*, *path=''*)
Convert to a gdal dataset.

to_rgb()

to_wgs84(*method='nneighbour'*)

transform(*proj*, *size=None*, *method='nneighbour'*)
A sample function to reproject and resample a GDAL dataset from within Python. The idea here is to reproject from one system to another, as well as to change the pixel size. The procedure is slightly long-winded, but goes like this:

1.Set up the two Spatial Reference systems.

2.Open the original dataset, and get the geotransform

3.Calculate bounds of new geotransform by projecting the UL corners

4.Calculate the number of pixels with the new projection & spacing

5.Create an in-memory raster dataset

6.Perform the projection

class pyspatial.raster.**RasterBase**(*RasterXSize*, *RasterYSize*, *geo_transform*, *proj*)
Bases: object

Provides methods and attributes common to both RasterBand and RasterDataset, particularly for converting shapes to pixels in the raster coordinate space. Stores a coordinate system for a raster.

Parameters

- **RasterYSize** (*RasterXSize,*) – Number of pixels in the width and height respectively.
- **geo_transform** (*list of float*) – GDAL coefficients for GeoTransform (defines boundaries and pixel size for a raster in lat/lon space).
- **proj** (*osr.SpatialReference*) – The spatial projection for the raster.

xsize, ysize
int

Number of pixels in the width and height respectively.

geo_transform
list of float

GDAL coefficients for GeoTransform (defines boundaries and pixel size for a raster in lat/lon space).

min_lon
float

The minimum longitude in proj coordinates

min_lat
float

The minimum latitude in proj coordinates

max_lat
float

The maximum latitude in proj coordinates

lon_px_size
float

Horizontal size of the pixel

lat_px_size
float

Vertical size of the pixel

proj
osr.SpatialReference

The spatial projection for the raster.

GetGeoTransform()
Returns affine transform from GDAL for describing the relationship between raster positions (in pixel/line coordinates) and georeferenced coordinates.

Returns

- **min_lon** (*float*) – The minimum longitude in raster coordinates.
- **lon_px_size** (*float*) – Horizontal size of each pixel.
- **geo_transform[2]** (*float*) – Not used in our case. In general, this would be used if the coordinate system had rotation or shearing.
- **max_lat** (*float*) – The maximum latitude in raster coordinates.
- **lat_px_size** (*float*) – Vertical size of the pixel.
- **geo_transform[5]** (*float*) – Not used in our case. In general, this would be used if the coordinate system had rotation or shearing.

References

http://www.gdal.org/gdal_tutorial.html

bbox()
Returns bounding box of raster in raster coordinates.

Returns

Bounding box in raster coordinates: (xmin : float

minimum longitude (leftmost)

ymin [float] minimum latitude (bottom)

xmax [float] maximum longitude (rightmost)

ymax [float] maximum latitude (top))

Return type ogr.Geometry

get_extent()
Returns extent in raster coordinates.

Returns

- **xmin** (*float*) – Minimum x-value (lon) of extent in raster coordinates.
- **xmax** (*float*) – Maximum x-value (lon) of extent in raster coordinates.
- **ymin** (*float*) – Minimum y-value (lat) of extent in raster coordinates.
- **ymax** (*float*) – Maximum y-value (lat) of extent in raster coordinates.

shape_to_pixel(*geom*)
Takes a feature and returns a shapely object transformed into the pixel coords.

Parameters **feat** (*osgeo.ogr.Geometry*) – Feature to be transformed.

Returns Feature in pixel coordinates.

Return type shapely.Polygon

to_geometry_grid(*minx*, *miny*, *maxx*, *maxy*)
Convert pixels into a geometry grid. All values should be in pixel cooridnates.

Returns

- *VectorLayer with index a tuple of the upper left corner coordinate*
- *of each pixel.*

to_pixels(*vector_layer*)
Takes a vector layer and returns list of shapely geometry transformed in pixel coordinates. If the projection of the vector_layer is different than the raster band projection, it transforms the coordinates first to raster projection.

Parameters **vector_layer** (VectorLayer) – Shapes to be transformed.

Returns Shapes in pixel coordinates.

Return type list of shapely.Polygon

to_raster_coord(*pxx*, *pxy*)
Convert pixel corrdinates -> raster coordinates

class pyspatial.raster.**RasterDataset**(*path_or_ds*, *xsize*, *ysize*, *geo_transform*, *proj*, *tile_regex=<_sre.SRE_Pattern object>*, *grid_size=None*, *index=None*, *tile_structure=None*, *tms_z=None*)
Bases: *pyspatial.raster.RasterBase*

Raster representation that supports tiled and untiled datasets, and allows querying with a set of shapes.

Raster may be tiled or untiled. A RasterDataset object may be queried one or multiple times with a set of shapes (in a VectorLayer). We also try to match the attribute and method names of gdal.Dataset when possible to allow for easy porting of caller code to use this class instead of gdal.Dataset, as this class transparently works on an untiled gdal.Dataset, in addition to the added functionality to handle tiled datasets.

Parameters

- **path_or_ds** (*str or gdal.Dataset*) –
- **xsize** (*int*) – The number of pixels in the X coordinate
- **ysize** (*int*) – The number of pixels in the Y coordinate
- **geo_transform** (*tuple*) –

- **proj** (*osr.SpatialReference*) – The spa
- **grid_size** (*int (default=None)*) – Number of pixels for each tile. Assumes that each tile is square.
- **index** (*dict (default=None)*) – Dictionary matching the Geojson spec describing boundary of each file. Typically generated by gdaltindex.
- **tile_structure** (*str (default="%d_%d.tif")*) – A string describing the file structure and the format of the tiles. In case of use of gdal2tiles tiles, this must be set to "%d/%d.png".

path

str

Path to raster data files.

grid_size

int

Number of pixels in width or height of each grid tile. If set to None, that indicates this is an untiled raster.

raster_bands

RasterBand, or

dict of (list of int): RasterBand

Dictionary storing raster arrays that have been read from disk. If untiled, this is set at initialization to the whole raster. If tiled, these are read in lazily as needed. Index is (x_grid, y_grid) where x_grid is x coordinate of leftmost pixel in this tile relative to minLon (and is a multiple of grid_size), and y_grid is y coordinate of uppermost pixel in this tile relative to maxLat (and is also a multiple of grid_size). See notes below for more information on how data is represented here.

shapes_in_tiles

dict of (int, int): set of str

What shapes are left to be processed in each tile. Key is (minx, maxy) of tile (upper left corner), and value is set of ids of shapes. This is initially set in query(), and shape ids are removed from this data structure for a tile once they have been processed. Tiles can be cleared from memory when there are no shapes left in their set.

Notes

Raster representation (tiled and untiled): The core functionality of this class is to look up pixel values (for a shape or set of shapes) in the raster. To do this, we store the raster in a 2D-array of pixels relative to (0,0) being the upper left corner aka (min_lon, max_lat) in lon/lat coordinates. We can then convert vector shapes into pixel space, and look up their values directly in the raster array. For an untiled raster, we can read in the raster directly during initialization.

Tiled representation: For a tiled dataset (ie. the data is split into multiple files), we still treat the overall raster upper left corner as (0,0), and recognize that each tile has a position relative to the overall raster pixel array. We store each tile in a 2D array in a dictionary keyed by the tile position relative to the overall raster position in pixel space. For example, a pixel at (118, 243) in a tiled dataset with grid size = 100 would be stored in raster_bands[(100, 200)][18][43]. As a memory utilization and performance enhancement, we lazily read tiles from disk when they are first needed and store them in raster_bands{} (for the lifetime of the RasterDataset object). If memory turns out to be a problem, it might make sense to store these in a LRU cache instead.

•Added band number

•Add support for color tables and raster attributes

get_values_for_pixels(*pxs*)
Look up values for a list of pixels in raster space.

Parameters **pxs** (*np.array*) – Array of pixel coordinates. Each row is [x_coord, y_coord] for one point.

Returns List of values in raster at pixel coordinates specified in pxs. Type is determined by GDAL2NP_CONVERSION from RasterBand data type.

Return type list of dtype

query(*vector_layer*, *ext_outline=False*, *ext_fill=True*, *int_outline=False*, *int_fill=False*, *scale_factor=4*, *missing_first=False*, *small_polygon_pixels=4*)
Query the dataset with a set of shapes (in a VectorLayer). The vectors will be reprojected into the projection of the raster. Any shapes in the vector layer that are not within the bounds of the raster will return with values and weights as np.array([]).

Parameters

- **vector_layer** (VectorLayer) – Set of shapes in vector format, with ids attached to each.
- **ext_outline** (*boolean (default False)*) – Include the outline of the shape in the raster
- **ext_fill** (*boolean (default True)*) – Fill the shape in the raster
- **int_outline** (*booelan (default False)*) – Include the outline of the interior shapes
- **int_fill** (*boolean (default False):*) – Fill the interior shapes
- **scale_factor** (*int (default 4)*) – The amount to scale the shape in X, Y before downscaling. The higher this number, the more precise the estimate of the overlap.
- **missing_first** (*boolean (default false)*) – Where the missing values should be at the beginning or the end of the results.
- **small_polygon_pixels** (*integer (default 4)*) – Number of pixels for the intersection of the polygon with the raster to be considered "small". This is a slow step that computes the exact intersection between the polygon and the raster in the cooridate space of the raster (not pixel space!).

Yields

- **RasterQueryResult. This is 3 attributes** (*id, values, weights. The*)
- *values are the pixel values from the raster. the weights are the fraction*
- *of the pixel that is occupied by the polgon.*

class pyspatial.raster.**RasterQueryResult**(*id*, *coordinates*, *values*, *weights*)
Container class to hold the result of a raster query.

id
str or int

The id of the shape in the vector layer

coordinates
np.ndarray

The requested raster pixel coordinates

values
np.ndarray

The values of the intersected pixels in the raster

weights
np.ndarray

The fraction of the polygon intersecting with the pixel

class pyspatial.raster.**TiledWebRaster**(*path*, *zoom*, *tile_size=256*, *bands=None*, *xy_tile_path='%s/%s.png'*)
Bases: *pyspatial.raster.RasterDataset*

Raster representation for tiled data sets commonly used on the web (e.g OpenLayers, GoogleMaps, etc.). These have been assumed to be produced using the gdal2tiles.py script (found in /scripts dir).

Assumes that the tiles are projected in Popular Visualisation CRS / Mercator (EPSG:3785)

Parameters

- **path** (*str*) – The path to the tiled data
- **zoom** (*int*) – Zoom to use, typically a value from 6 to 15.
- **tile_size** (*int*) – n x n size of each tile in pixels, default is 256
- **bands** (*list of ints*) – The bands to use. Default is [1, 2, 3]
- **xy_tile_path** (*str*) – The tile path structure, default="%s/%s.png"

path
str

Path to raster data files.

resoltution
float

Number of meters in both x & y that each pixel represents. For example, at a zoom of 11, each pixel is approx 76 m x 76 m.

Notes

Since these datasets are typically png files, there will be a substantial performance hit due to the CPU overhead of decompression

get_values_for_pixels(*pxs*)
Look up values for a list of pixels in raster space.

Parameters **pxs** (*np.array*) – Array of pixel coordinates. Each row is [x_coord, y_coord] for one point.

Returns List of values in raster at pixel coordinates specified in pxs. Type is determined by GDAL2NP_CONVERSION from RasterBand data type.

Return type list of dtype

pyspatial.raster.**rasterize**(*shp*, *ext_outline=False*, *ext_fill=True*, *int_outline=False*, *int_fill=False*, *scale_factor=4*)
Convert a vector shape to a raster. Assumes the shape has already been transformed in to a pixel based coordinate system. The algorithm checks for the intersection of each point in the shape with a pixel grid created by the bounds of the shape. Partial overlaps are estimated by scaling the image in X and Y by the scale factor, rasterizing the shape, and downscaling (using mean), back to the bounds of the original shape.

Parameters

- **shp** (*shapely.Polygon or Multipolygon*) – The shape to rasterize
- **ext_outline** (*boolean (default False)*) – Include the outline of the shape in the raster
- **ext_fill** (*boolean (default True)*) – Fill the shape in the raster
- **int_outline** (*booelan (default False)*) – Include the outline of the interior shapes
- **int_fill** (*boolean (default False):*) – Fill the interior shapes
- **scale_factor** (*int (default 4)*) – The amount to scale the shape in X, Y before downscaling. The higher this number, the more precise the estimate of the overlap.

Returns

Return type np.ndarray representing the rasterized shape.

pyspatial.raster.**read_band**(*path*, *band_number=1*)

Read a single band from a raster into memory.

Parameters

- **path** (*string*) – Path to the raster file. Can be either local or s3/gs.
- **band_number** (*int*) – The band number to use

Returns

Return type *RasterBand*

pyspatial.raster.**read_catalog**(*dataset_catalog_filename_or_handle*, *workdir=None*)

Take a catalog file and create a raster dataset

Parameters **dataset_catalog_filename_or_handle** (*str or opened file handle*) – if str : Path to catalog file for the dataset. May be relative or absolute.

Catalog files are in json format, and usually represent a type of data (e.g. CDL) and a year (e.g. 2014).

Returns

Return type *RasterDataset*

See also:

scripts()

raster_query_test.py() Simple examples of exercising RasterQuery on tiled and untiled datasets, and computing stats from results.

vector.py() Details of VectorLayer.

pyspatial.raster.**read_raster**(*path*, *band_number=1*)

Create a raster dataset from a single raster file

Parameters

- **path** (*string*) – Path to the raster file. Can be either local or s3/gs.
- **band_number** (*int*) – The band number to use

Returns

Return type *RasterDataset*

pyspatial.raster.**read_vsimem**(*path*, *band_number=1*)

Read a single band into memory from a raster. This method does not support all raster formats, only those that are supported by /vsimem

Parameters

- **path** (*string*) – Path to the raster file. Can be either local or s3/gs.
- **band_number** (*int*) – The band number to use

Returns

Return type *RasterBand*

4.1.8 pyspatial.spatiallib module

4.1.9 pyspatial.utils module

pyspatial.utils.**get_projection**(*obj*, *layer_index=0*)

pyspatial.utils.**projection_from_epsg**(*epsg=4326*)

Returns a projection from an epsg integer. If no argument is provided, uses 4326.

pyspatial.utils.**projection_from_string**(*proj_str='+proj=longlat +ellps=WGS84 +datum=WGS84 +no_defs'*)

Returns a projection from a Proj4 string. If no argument is provided, uses WGS84/EPSG:4326

pyspatial.utils.**projection_from_wkt**(*wkt=<MagicMock name='mock.SRS_WKT_WGS84' id='139917863948496'>*)

Returns a projection from well known text. If no argument is provided, uses 4326 equivalent

pyspatial.utils.**svg_line**(*shp*, *scale_factor*, *stroke*)

Returns SVG polyline element for the LineString geometry. :param scale_factor: Multiplication factor for the SVG stroke-width. Default is 1. :type scale_factor: float :param stroke_color: Hex string for stroke color. Default is to use "#66cc99" if

geometry is valid, and "#ff3333" if invalid.

pyspatial.utils.**svg_multiline**(*shp*, *scale_factor*, *stroke*)

Returns a group of SVG polyline elements for the LineString geometry. :param scale_factor: Multiplication factor for the SVG stroke-width. Default is 1. :type scale_factor: float :param stroke_color: Hex string for stroke color. Default is to use "#66cc99" if

geometry is valid, and "#ff3333" if invalid.

pyspatial.utils.**svg_multipolygon**(*shp*, *scale_factor*, *fill*)

Returns group of SVG path elements for the MultiPolygon geometry. :param scale_factor: Multiplication factor for the SVG stroke-width. Default is 1. :type scale_factor: float :param fill_color: Hex string for fill color. Default is to use "#66cc99" if

geometry is valid, and "#ff3333" if invalid.

pyspatial.utils.**svg_polygon**(*shp*, *scale_factor*, *fill*)
Returns SVG path element for the Polygon geometry. :param scale_factor: Multiplication factor for the SVG stroke-width. Default is 1. :type scale_factor: float :param fill_color: Hex string for fill color. Default is to use "#66cc99" if

geometry is valid, and "#ff3333" if invalid.

pyspatial.utils.**to_svg**(*shp*, *color='#1f78b4'*)
Function that takes a shapely object and returns an svg of that object. Currently only supports (Multi)LineString, and (Multi)Polygon.

4.1.10 pyspatial.vector module

class pyspatial.vector.**VectorLayer**(**args*, ***kwargs*)
Bases: pandas.core.series.Series

Parameters

- **geometries** (*org.Feature[], ogr.Geometry[], shapely.BaseGeometry[]*) –
- **proj** (*osr.SpatialReference*) – The projection for the geometries. Defaults to EPSG:4326.
- **index** (*iterable*) – The index to use for the shapes

_sindex
rtree.index.Index

The spatial index. Initially None, but can be built with build_sindex()

append(**args*, ***kwargs*)

areas(*proj=None*)
Compute the areas for each of the shapes in the vector layer.

Parameters **proj** (*string or osr.SpatialReference (default=None)*) – valid strings are 'albers' or 'utm'. If None, no transformation of coordinates.

Returns

Return type pandas.Series

Note: 'utm' should only be used for small polygons when centimeter level accuraccy is needed. Othewise the area will be incorrect. Similar issues can happen when polygons cross utm boundaries.

bbox()
Return a geometry representing the bounding box of the layer

boundingboxes()
Return a VectorLayer with the bounding boxes of each geometry

build_sindex()

centroids(*format='VectorLayer'*)
Get a DataFrame with "x" and "y" columns for the centroid of each feature.

Parameters **format** (*str (default='VectorLayer')*) – Return type of the centroids. available options are 'Series', 'DataFrame', or 'VectorLayer'. 'Series' will a collection of (x, y) tuples. 'DataFrame' will be a DataFrame with columns 'x' and 'y'

contains(*shp*, *index_only=False*)

Return a vector layer with only those shapes in the vector layer that contain shp

Parameters

- **shp** (*shapely.BaseGeometry, ogr.Geometry, or ogr.Feature*) – The shape to test against
- **index_only** (*boolean (default False)*) – Return the ids only (not a new vector layer)

See also:

http() //toblerity.org/shapely/manual.html#object.contains

crosses(*shp*, *index_only=False*)

Return a vector layer with only those shapes in the vector layer that cross shp

Parameters

- **shp** (*shapely.BaseGeometry, ogr.Geometry, or ogr.Feature*) – The shape to test against
- **index_only** (*boolean (default False)*) – Return the ids only (not a new vector layer)

See also:

http() //toblerity.org/shapely/manual.html#object.crosses

difference(*shp*, *kind='left'*)

Cut the shapes in the VectorLayer to match the difference specified by shp.

Parameters

- **shp** (*shapely geometry or ogr Feature/Geometry*) –
- **kind** (*str*) – Either "left", "right", or "symmetric". In the case of "left" take geom.Difference(shp). In the case of "right", take shp.Difference(geom). Where geom is each geometry in the VectorLayer

Returns

Return type *VectorLayer*

disjoint(*shp*, *index_only=False*)

Return a vector layer with only those shapes in the vector layer that are disjoint with shp

Parameters

- **shp** (*shapely.BaseGeometry, ogr.Geometry, or ogr.Feature*) – The shape to test against
- **index_only** (*boolean (default False)*) – Return the ids only (not a new vector layer)

See also:

http() //toblerity.org/shapely/manual.html#object.disjoint

distances(*shp*, *proj=None*)
Compute the euclidean distances for each of the shapes in the vector layer. If proj is not none, it will transform shp into proj.

Note: if shp is a shapely object, it is upto to the user to make sure shp is in the correct coordinate system.

Parameters **proj** (*string or osr.SpatialReference (default=None)*) – valid strings are 'albers' or 'utm'. If None, no transformation of coordinates.

Returns

Return type pandas.Series

Note: 'utm' should only be used for small polygons when centimeter level accuraccy is needed. Othewise the area will be incorrect. Similar issues can happen when polygons cross utm boundaries.

---

envelopes()
The the envelope of each shape as xmin, xmax, ymin, ymax. Returns a pandas.Series.

equals(*shp*, *index_only=False*)
Return a vector layer with only those shapes in the vector layer that are equal shp

Parameters

- **shp** (*shapely.BaseGeometry, ogr.Geometry, or ogr.Feature*) – The shape to test against
- **index_only** (*boolean (default False)*) – Return the ids only (not a new vector layer)

See also:

http() //toblerity.org/shapely/manual.html#binary-predicates

features

filter_by_id(*ids*)
Return a vector layer with only those shapes with id in ids.

Parameters **ids** (*iterable*) – The ids to filter on

get_extent()
The xmin, xmax, ymin, ymax values of the layer

icontains(*shp*)
Return an index with only those shapes in the vector layer that contain shp

Parameters **shp** (*shapely.BaseGeometry, ogr.Geometry, or ogr.Feature*) – The shape to test against

Returns

Return type pandas.Index

icrosses(*shp*)
Return an index with only those shapes in the vector layer that crosses shp

Parameters **shp** (*shapely.BaseGeometry, ogr.Geometry, or ogr.Feature*) – The shape to test against

Returns

Return type pandas.Index

idisjoint(*shp*)
Return an index with only those shapes in the vector layer that is disjoint to shp

Parameters **shp** (*shapely.BaseGeometry, ogr.Geometry, or ogr.Feature*) – The shape to test against

Returns

Return type pandas.Index

ids

iequals(*shp*)
Return an index with only those shapes in the vector layer that equals shp

Parameters **shp** (*shapely.BaseGeometry, ogr.Geometry, or ogr.Feature*) – The shape to test against

Returns

Return type pandas.Index

iintersects(*shp*)
Return an index with only those shapes in the vector layer that intersect with shp

Parameters **shp** (*shapely.BaseGeometry, ogr.Geometry, or ogr.Feature*) – The shape to test against

Returns

Return type pandas.Index

intersection(*shp*)
Cut the shapes in the VectorLayer to match the intersection specified by shp.

Parameters **shp** (*shapely geometry or ogr Feature/Geometry*) –

Returns

Return type VectorLayer interesected by shp

intersects(*shp*, *index_only=False*)
Return a vector layer with only those shapes in the vector layer that intersect with shp

Parameters

- **shp** (*shapely.BaseGeometry, ogr.Geometry, or ogr.Feature*) – The shape to test against
- **index_only** (*boolean (default False)*) – Return the ids only (not a new vector layer)

See also:

http() //toblerity.org/shapely/manual.html#object.intersects

is_empty(*index_only=False*)
Get vector layer with the empty shapes

is_invalid(*index_only=False*)
Get vector layer with invalid shapes.

is_ring(*index_only=False*)
Get vector layer with the ring shapes

is_valid(*index_only=False*)
Get vector layer with valid shapes.

items()

itouches(*shp*)
Return an index with only those shapes in the vector layer that touches shp

Parameters **shp** (*shapely.BaseGeometry, ogr.Geometry, or ogr.Feature*) – The shape to test against

Returns

Return type pandas.Index

iwithin(*shp*)
Return an index with only those shapes in the vector layer that are within shp

Parameters **shp** (*shapely.BaseGeometry, ogr.Geometry, or ogr.Feature*) – The shape to test against

Returns

Return type pandas.Index

map(*f*, *as_geometry=False*)
Apply a function, f, over all the geometries.

Returns

Return type pandas.Series(as_geometry=False) or VectorLayer(as_geometry=True)

nearest(*shp*, *max_neighbors=5*)

select(**args*, ***kwargs*)

size_bytes()
Get the size of the geometry in bytes

sort(*kind='upper_left_corners', columns=['y', 'x'], ascending=True, index_only=False*)
Sort the vector layer by upper_left_corners or centroids

Parameters

- **kind** (*str*) – Either "upper_left_corners" or "centroids"
- **columns** (*list of str (default ["y", "x"]*) – Order in which to sort the shapes (ie. sort primarily by y-axis or x-axis). Will be passed to pandas.DataFrame.sort.
- **ascending** (*boolean (default True)*) – Sort by columns in ascending or descending order.

Returns Shape ids sorted by columns.

Return type list

sort_index(**args*, ***kwargs*)

symmetric_difference(*shp*, *kind='left'*)
Cut the shapes in the VectorLayer to match the symmetric difference specified by shp.

Parameters **shp** (*shapely geometry or ogr Feature/Geometry*) –

Returns

Return type *VectorLayer*

take (**args*, ***kwargs*)

to_dict (*df=None*)
Return a dictionary representation of the object. Based off the GeoJSON spec. Will transform the vector layer into WGS84 (EPSG:4326).

Parameters **df** (*pandas.DataFrame (default=None)*) – The dataframe to supply the properties of the features. The index of df must match the ids of the VectorLayer.

Returns

Return type dict

to_geometry (*ids=None*, *proj=None*)

to_json (*path=None*, *df=None*, *precision=6*)
Return the layer as a GeoJSON. If a path is provided, it will save to the path. Otherwise, will return a string.

Parameters

- **path** (*str, (default=None)*) – The path to save the geojson data.
- **df** (*pandas.DataFrame (default=None)*) – The dataframe to supply the properties of the features. The index of df must match the ids of the VectorLayer.
- **precision** (*int*) – Number of decimal places to keep for floats

Returns

Return type geojson string

to_shapefile (*path*, *df=None*)
Write the VectorLayer to an ESRI Shapefile.

Only supports simple types for the attributes (int, float, str). Any columns in the df that are not simple types will be ignored.

Parameters

- **path** (*str*) – Path to where you want to save the file. Can be local or s3/gs.
- **df** (*pandas.DataFrame (default=None)*) – Attach the attributes to the vector layer. Similar to to_dict.

to_shapely (*ids=None*)

to_svg (*ids=None*, *ipython=False*)
Return svg represention. Can output in IPython friendly form. If ids is None, will return a pandas.Series with all shapes as svg.

Parameters

- **ids** (*str or iterable (default=None)*) – The values of the geometries to convert to svg. If string, returns a string, if iterable, returns a Series.
- **ipython** (*(default=False)*) – Render in IPython friendly format

Returns

Return type str or pandas.Series

to_wgs84 ()
Transform the VectorLayer into WGS84

touches (*shp*, *index_only=False*)
Return a vector layer with only those shapes in the vector layer that touches shp

Parameters

- **shp** (*shapely.BaseGeometry, ogr.Geometry, or ogr.Feature*) – The shape to test against
- **index_only** (*boolean (default False)*) – Return the ids only (not a new vector layer)

See also:

http() //toblerity.org/shapely/manual.html#object.touches

transform(*target_proj*)

unary_union()

union(*shp*)

Cut the shapes in the VectorLayer to match the union specified by shp.

Parameters **shp** (*shapely geometry or ogr Feature/Geometry*) –

Returns

Return type VectorLayer interesected by shp

upper_left_corners()

Get a DataFrame with "x" and "y" columns for the min_lon, max_lat of each feature

within(*shp*, *index_only=False*)

Return a vector layer with only those shapes in the vector layer that are within shp.

Parameters

- **shp** (*shapely.BaseGeometry, ogr.Geometry, or ogr.Feature*) – The shape to test against
- **index_only** (*boolean (default False)*) – Return the ids only (not a new vector layer)

See also:

http() //toblerity.org/shapely/manual.html#object.within

pyspatial.vector.**bounding_box**(*envelope*, *proj*)

pyspatial.vector.**fetch_geojson**(*path*)

pyspatial.vector.**from_series**(*geom_series*, *proj=None*)

Create a VectorLayer from a pandas.Series object. If the geometries do not have an spatial reference, EPSG:4326 is assumed.

Parameters

- **geom_series** (*pandas.Series*) – The series object with shapely geometries
- **proj** (*osr.SpatialReference*) – The projection to use, defaults to EPSG:4326

Returns

Return type *VectorLayer*

pyspatial.vector.**read_datasource**(*ds*, *layer=0*, *index=None*)

pyspatial.vector.**read_geojson**(*path_or_str*, *index=None*)

Create a vector layer from a geojson object. Assumes that the data has a projection of EPSG:4326

Parameters

- **path_or_str** (*string*) – path or json string
- **index** (*string or iterable*) – If string, the column in the "properties" of each feature to use as the index. If iterable, use the iterable as the index.

Returns

Return type Tuple of (VectorLayer, pandas.DataFrame of properties)

pyspatial.vector.**read_layer**(*path*, *layer=0*, *index=None*)
Create a vector layer from the specified path. Will try to read using ogr.OpenShared.

Parameters

- **path_or_str** (*string*) – path or json string
- **layer** (*integer*) – The layer number to use. Use ogrinfo to see the available layers.
- **index** (*string or iterable (default=None)*) – If string, the column in the "properties" of each feature to use as the index. If iterable, use the iterable as the index. If not specified, will create an integer based index.

Returns

Return type Tuple of (VectorLayer, pandas.DataFrame of properties)

pyspatial.vector.**set_theoretic_methods**(*function*, *shp1*, *shp2*)

pyspatial.vector.**to_feature**(*shp*, *fid*, *proj=None*)

pyspatial.vector.**to_geometry**(*shp*, *copy=False*, *proj=None*)
Convert shp to a ogr.Geometry.

Parameters

- **shp** (*ogr.Geometry, ogr.Feature, or shapely.BaseGeometry*) – The shape you want to convert
- **copy** (*boolean (default=False)*) – Return a copy of the shape instead of a reference
- **proj** (*str or osr.SpatialReference (default=None)*) – The projection of the shape to define (if the shape is not projection aware), or transform to (if projection aware). If a string is provided, it assumes that it is in PROJ4.

Returns

Return type ogr.Geometry

pyspatial.vector.**to_shapely**(*feat*, *proj=None*)

4.1.11 pyspatial.visualize module

class pyspatial.visualize.**HTMLMap**(*lat*, *lng*, *zoom=5*, *data=None*, *info_cols=None*, *height='100%'*, *static_js='https://granular-labs.s3.amazonaws.com/static/js'*, *static_css='https://granular-labs.s3.amazonaws.com/static/css'*, *static_img='https://granular-labs.s3.amazonaws.com/static/img'*)
Bases: object

add_markers(*name*, *shapes*, *style=None*, *text=None*)

add_shapes(*name*, *shapes*, *style=None*, *text=None*)
Add shapes to the map. Accepts pandas Series or list of shapely or ogr.Geometry objects, or a dictionary or string matching the geojson spec.

add_text(*name*, *points*, *values*, *style=None*)
Not implemented yet

choropleth(*column=None*, *levels=6*, *palette='Reds'*)
Create a choropleth.

#TODO: - Add style per choropleth. - Add support for different discretization schemes

Parameters

- **column** (*str*) – The column to use for the choropleth. Must exist in the data provided.
- **levels** (*int, default None*) – Number of levels to uses for the scale. The allowed amount varies based on the palette that is chosen.
- **palette** (*string or dict, default will use 'Reds'*) – Uses color brewer (http://colorbrewer2.org/). If you pass a dict, the keys are the string values in the column, and the values are the html hex color for that value.

render_ipython(*height='500px'*, *width='100%'*)

save(*path*)
Save the html to a file. Can be local or s3. If s3, assumes that a .boto file exists

set_baselayer(*geo_data*)

pyspatial.visualize.**get_geojson_dict**(*geo_data*)

pyspatial.visualize.**get_latlngs**(*geo_data*)

pyspatial.visualize.**to_feature**(*shp*, *_id*)

pyspatial.visualize.**to_latlng**(*shp*)

4.1.12 Module contents

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

.

p

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

Z