Python Shapefile Library

Author: Joel Lawhead <jlawhead@geospatialpython.com>
Revised: June 23, 2013

Contents

Overview

The Python Shapefile Library (pyshp) provides read and write support for the Esri Shapefile format. The Shapefile format is a popular Geographic Information System vector data format created by Esri. For more information about this format please read the well-written "ESRI Shapefile Technical Description - July 1998" located at http://www.esri.com/library/whitepapers/pdfs/shapefile.pdf. The Esri document describes the shp and shx file formats. However a third file format called dbf is also required. This format is documented on the web as the "XBase File Format Description" and is a simple file-based database format created in the 1960's. For more on this specification see: http://www.clicketyclick.dk/databases/xbase/format/index.html

Both the Esri and XBase file-formats are very simple in design and memory efficient which is part of the reason the shapefile format remains popular despite the numerous ways to store and exchange GIS data available today.

Pyshp is compatible with Python 2.4-3.x.

This document provides examples for using pyshp to read and write shapefiles.

Currently the sample census blockgroup shapefile referenced in the examples is only available on the google code project site at http://code.google.com/p/pyshp. These examples are straight-forward and you can also easily run them against your own shapefiles manually with minimal modification. Other examples for specific topics are continually added to the pyshp wiki on google code and the blog http://GeospatialPython.com.

Important: For information about map projections, shapefiles, and Python please visit: http://code.google.com/p/pyshp/wiki/MapProjections

I sincerely hope this library eliminates the mundane distraction of simply reading and writing data, and allows you to focus on the challenging and FUN part of your geospatial project.

Examples

Before doing anything you must import the library.

```
>>> import shapefile
```

The examples below will use a shapefile created from the U.S. Census Bureau Blockgroups data set near San Francisco, CA and available in the subversion repository of the pyshp google code site.

Reading Shapefiles

To read a shapefile create a new "Reader" object and pass it the name of an existing shapefile. The shapefile format is acutally a collection of three files. You specify the base filename of the shapefile or the complete filename of any of the shapefile component files.

```
>>> sf = shapefile.Reader("shapefiles/blockgroups")
```

OR

```
>>> sf = shapefile.Reader("shapefiles/blockgroups.shp")
```

OR

```
>>> sf = shapefile.Reader("shapefiles/blockgroups.dbf")
```

OR any of the other 5+ formats which are potentially part of a shapefile. The library does not care about extensions.

Reading Shapefiles from File-Like Objects

You can also load shapefiles from any Python file-like object using keyword arguments to specify any of the three files. This feature is very powerful and allows you to load shapefiles from a url, from a zip file, serialized object, or in some cases a database.

```
>>> myshp = open("shapefiles/blockgroups.shp", "rb")
>>> mydbf = open("shapefiles/blockgroups.dbf", "rb")
>>> r = shapefile.Reader(shp=myshp, dbf=mydbf)
```

Notice in the examples above the shx file is never used. The shx file is a very simple fixed-record index for the variable length records in the shp file. This file is optional for reading. If it's available pyshp will use the shx file to access shape records a little faster but will do just fine without it.

Reading Geometry

A shapefile's geometry is the collection of points or shapes made from verticies and implied arcs representing physical locations. All types of shapefiles just store points. The metadata about the points determine how they are handled by software.

You can get the a list of the shapefile's geometry by calling the shapes() method.

```
>>> shapes = sf.shapes()
```

The shapes method returns a list of Shape objects describing the geometry of each shape record.

```
>>> len(shapes)
663
```

You can iterate through the shapefile's geometry using the iterShapes() method.

```
>>> len(list(sf.iterShapes()))
663
```

Each shape record contains the following attributes:

```
>>> for name in dir(shapes[3]):
...     if not name.startswith('__'):
...         name
'bbox'
'parts'
'points'
'shapeType'
```

- shapeType: an integer representing the type of shape as defined by the shapefile specification.

```
>>> shapes[3].shapeType
5
```

- bbox: If the shape type contains multiple points this tuple describes the lower left (x,y) coordinate and upper right corner coordinate creating a complete box around the points. If the shapeType is a Null (shapeType == 0) then an AttributeError is raised.

```
>>> # Get the bounding box of the 4th shape.
>>> # Round coordinates to 3 decimal places
>>> bbox = shapes[3].bbox
>>> ['%.3f' % coord for coord in bbox]
['-122.486', '37.787', '-122.446', '37.811']
```

- parts: Parts simply group collections of points into shapes. If the shape record has multiple parts this attribute contains the index of the first point of each part. If there is only one part then a list containing 0 is returned.

```
>>> shapes[3].parts
[0]
```

- points: The points attribute contains a list of tuples containing an (x,y) coordinate for each point in the shape.

```
>>> len(shapes[3].points)
173
```

```
>>> # Get the 8th point of the fourth shape
>>> # Truncate coordinates to 3 decimal places
>>> shape = shapes[3].points[7]
>>> ['%.3f' % coord for coord in shape]
['-122.471', '37.787']
```

To read a single shape by calling its index use the shape() method. The index is the shape's count from 0. So to read the 8th shape record you would use its index which is 7.

```
>>> s = sf.shape(7)
```

```
>>> # Read the bbox of the 8th shape to verify
>>> # Round coordinates to 3 decimal places
>>> ['%.3f' % coord for coord in s.bbox]
['-122.450', '37.801', '-122.442', '37.808']
```

Reading Records

A record in a shapefile contains the attributes for each shape in the collection of geometry. Records are stored in the dbf file. The link between geometry and attributes is the foundation of Geographic Information Systems. This critical link is implied by the order of shapes and corresponding records in the shp geometry file and the dbf attribute file.

The field names of a shapefile are available as soon as you read a shapefile. You can call the "fields" attribute of the shapefile as a Python list. Each field is a Python list with the following information:

- Field name: the name describing the data at this column index.
- Field type: the type of data at this column index. Types can be: Character, Numbers, Longs, Dates, or Memo. The "Memo" type has no meaning within a GIS and is part of the xbase spec instead.
- Field length: the length of the data found at this column index. Older GIS software may truncate this length to 8 or 11 characters for "Character" fields.
- Decimal length: the number of decimal places found in "Number" fields.

To see the fields for the Reader object above (sf) call the "fields" attribute:

```
>>> fields = sf.fields
```

```
>>> assert fields == [("DeletionFlag", "C", 1, 0), ["AREA", "N", 18, 5],
... ["BKG_KEY", "C", 12, 0], ["POP1990", "N", 9, 0], ["POP90_SQMI", "N", 10, 1],
... ["HOUSEHOLDS", "N", 9, 0],
... ["MALES", "N", 9, 0], ["FEMALES", "N", 9, 0], ["WHITE", "N", 9, 0],
... ["BLACK", "N", 8, 0], ["AMERI_ES", "N", 7, 0], ["ASIAN_PI", "N", 8, 0],
... ["OTHER", "N", 8, 0], ["HISPANIC", "N", 8, 0], ["AGE_UNDER5", "N", 8, 0],
... ["AGE_5_17", "N", 8, 0], ["AGE_18_29", "N", 8, 0], ["AGE_30_49", "N", 8, 0],
... ["AGE_50_64", "N", 8, 0], ["AGE_65_UP", "N", 8, 0],
... ["NEVERMARRY", "N", 8, 0], ["MARRIED", "N", 9, 0], ["SEPARATED", "N", 7, 0],
... ["WIDOWED", "N", 8, 0], ["DIVORCED", "N", 8, 0], ["HSEHLD_1_M", "N", 8, 0],
... ["HSEHLD_1_F", "N", 8, 0], ["MARHH_CHD", "N", 8, 0],
... ["MARHH_NO_C", "N", 8, 0], ["MHH_CHILD", "N", 7, 0],
... ["FHH_CHILD", "N", 7, 0], ["HSE_UNITS", "N", 9, 0], ["VACANT", "N", 7, 0],
... ["OWNER_OCC", "N", 8, 0], ["RENTER_OCC", "N", 8, 0],
... ["MEDIAN_VAL", "N", 7, 0], ["MEDIANRENT", "N", 4, 0],
... ["UNITS_1DET", "N", 8, 0], ["UNITS_1ATT", "N", 7, 0], ["UNITS2", "N", 7, 0],
```

```
... ["UNITS3_9", "N", 8, 0], ["UNITS10_49", "N", 8, 0],
... ["UNITS50_UP", "N", 8, 0], ["MOBILEHOME", "N", 7, 0]]
```

You can get a list of the shapefile's records by calling the records() method:

```
>>> records = sf.records()
```

```
>>> len(records)
663
```

Similar to the geometry methods, you can iterate through dbf records using the recordsIter() method.

```
>>> len(list(sf.iterRecords()))
663
```

Each record is a list containing an attribute corresponding to each field in the field list.

For example in the 4th record of the blockgroups shapefile the 2nd and 3rd fields are the blockgroup id and the 1990 population count of that San Francisco blockgroup:

```
>>> records[3][1:3]
['060750601001', 4715]
```

To read a single record call the record() method with the record's index:

```
>>> rec = sf.record(3)
```

```
>>> rec[1:3]
['060750601001', 4715]
```

Reading Geometry and Records Simultaneously

You way want to examine both the geometry and the attributes for a record at the same time. The shapeRecord() and shapeRecords() method let you do just that.

Calling the shapeRecords() method will return the geometry and attributes for all shapes as a list of ShapeRecord objects. Each ShapeRecord instance has a "shape" and "record" attribute. The shape attribute is a ShapeRecord object as dicussed in the first section "Reading Geometry". The record attribute is a list of field values as demonstrated in the "Reading Records" section.

```
>>> shapeRecs = sf.shapeRecords()
```

Let's read the blockgroup key and the population for the 4th blockgroup: >>> shapeRecs[3].record[1:3] ['060750601001', 4715]

Now let's read the first two points for that same record:

```
>>> points = shapeRecs[3].shape.points[0:2]
```

```
>>> len(points)
2
```

The shapeRec() method reads a single shape/record pair at the specified index. To get the 4th shape record from the blockgroups shapfile use the third index:

```
>>> shapeRec = sf.shapeRecord(3)
```

The blockgroup key and population count:

```
>>> shapeRec.record[1:3]
['060750601001', 4715]
```

```
>>> points = shapeRec.shape.points[0:2]
```

```
>>> len(points)
2
```

Writing Shapefiles

The PSL tries to be as flexible as possible when writing shapefiles while maintaining some degree of automatic validation to make sure you don't accidentally write an invalid file.

The PSL can write just one of the component files such as the shp or dbf file without writing the others. So in addition to being a complete shapefile library, it can also be used as a basic dbf (xbase) library. Dbf files are a common database format which are often useful as a standalone simple database format. And even shp files occasionaly have uses as a standalone format. Some web-based GIS systems use an user-uploaded shp file to specify an area of interest. Many precision agriculture chemical field sprayers also use the shp format as a control file for the sprayer system (usually in combination with custom database file formats).

To create a shapefile you add geometry and/or attributes using methods in the Writer class until you are ready to save the file.

Create an instance of the Writer class to begin creating a shapefile:

```
>>> w = shapefile.Writer()
```

Setting the Shape Type

The shape type defines the type of geometry contained in the shapefile. All of the shapes must match the shape type setting.

Shape types are represented by numbers between 0 and 31 as defined by the shapefile specification. It is important to note that numbering system has several reserved numbers which have not been used yet therefore the numbers of the existing shape types are not sequential.

There are three ways to set the shape type: - Set it when creating the class instance. - Set it by assigning a value to an existing class instance. - Set it automatically to the type of the first shape by saving the shapefile.

To manually set the shape type for a Writer object when creating the Writer:

```
>>> w = shapefile.Writer(shapeType=1)
```

```
>>> w.shapeType
1
```

OR you can set it after the Writer is created:

```
>>> w.shapeType = 3
```

```
>>> w.shapeType
3
```

Geometry and Record Balancing

Because every shape must have a corresponding record it is critical that the number of records equals the number of shapes to create a valid shapefile. To help prevent accidental misalignment the PSL has an "auto balance" feature to make sure when you add either a shape or a record the two sides of the equation line up. This feature is NOT turned on by default. To activate it set the attribute autoBalance to 1 (True):

```
>>> w.autoBalance = 1
```

You also have the option of manually calling the balance() method each time you add a shape or a record to ensure the other side is up to date. When balancing is used null shapes are created on the geometry side or a record with a value of "NULL" for each field is created on the attribute side.

The balancing option gives you flexibility in how you build the shapefile.

Without auto balancing you can add geometry or records at anytime. You can create all of the shapes and then create all of the records or vice versa. You can use the balance method after creating a shape or record each time and make updates later. If you do not use the balance method and forget to manually balance the geometry and attributes the shapefile will be viewed as corrupt by most shapefile software.

With auto balanacing you can add either shapes or geometry and update blank entries on either side as needed. Even if you forget to update an entry the shapefile will still be valid and handled correctly by most shapefile software.

Adding Geometry

Geometry is added using one of three methods: "null", "point", or "poly". The "null" method is used for null shapes, "point" is used for point shapes, and "poly" is used for everything else.

Adding a Null shape

Because Null shape types (shape type 0) have no geometry the "null" method is called without any arguments.

```
>>> w = shapefile.Writer()
```

```
>>> w.null()
```

The writer object's shapes list will now have one null shape:

```
>>> assert w.shapes()[0].shapeType == shapefile.NULL
```

Adding a Point shape

Point shapes are added using the "point" method. A point is specified by an x, y, and optional z (elevation) and m (measure) value.

```
>>> w = shapefile.Writer()
```

```
>>> w.point(122, 37) # No elevation or measure values
```

```
>>> w.shapes()[0].points
[[122, 37, 0, 0]]
```

```
>>> w.point(118, 36, 4, 8)
```

```
>>> w.shapes()[1].points
[[118, 36, 4, 8]]
```

Adding a Poly shape

"Poly" shapes can be either polygons or lines. Shapefile polygons must have at least 4 points and the last point must be the same as the first. PyShp automatically enforces closed polygons. A line must have at least two points. Because of the similarities between these two shape types they are created using a single method called "poly".

```
>>> w = shapefile.Writer()
```

```
>>> w.poly(shapeType=3, parts=[[[122,37,4,9], [117,36,3,4]], [[115,32,8,8],
... [118,20,6,4], [113,24]]])
```

Creating Attributes

Creating attributes involves two steps. Step 1 is to create fields to contain attribute values and step 2 is to populate the fields with values for each shape record.

The following attempts to create a complete shapefile:

```
>>> w = shapefile.Writer(shapefile.POINT)
>>> w.point(1,1)
>>> w.point(3,1)
>>> w.point(4,3)
>>> w.point(2,2)
>>> w.field('FIRST_FLD')
>>> w.field('SECOND_FLD','C','40')
>>> w.record('First','Point')
>>> w.record('Second','Point')
>>> w.record('Third','Point')
>>> w.record('Fourth','Point')
>>> w.save('shapefiles/test/point')
```

```
>>> w = shapefile.Writer(shapefile.POLYGON)
>>> w.poly(parts=[[[1,5],[5,5],[5,1],[3,3],[1,1]]])
>>> w.field('FIRST_FLD','C','40')
>>> w.field('SECOND_FLD','C','40')
>>> w.record('First','Polygon')
>>> w.save('shapefiles/test/polygon')
```

```
>>> w = shapefile.Writer(shapefile.POLYLINE)
>>> w.line(parts=[[[1,5],[5,5],[5,1],[3,3],[1,1]]])
```

```
>>> w.poly(parts=[[[1,3],[5,3]]], shapeType=shapefile.POLYLINE)
>>> w.field('FIRST_FLD','C','40')
>>> w.field('SECOND_FLD','C','40')
>>> w.record('First','Line')
>>> w.record('Second','Line')
>>> w.save('shapefiles/test/line')
```

You can also add attributes using keyword arguments where the keys are field names.

```
>>> w = shapefile.Writer(shapefile.POLYLINE)
>>> w.line(parts=[[[1,5],[5,5],[5,1],[3,3],[1,1]]])
>>> w.field('FIRST_FLD','C','40')
>>> w.field('SECOND_FLD','C','40')
>>> w.record(FIRST_FLD='First', SECOND_FLD='Line')
>>> w.save('shapefiles/test/line')
```

File Names

File extensions are optional when reading or writing shapfiles. If you specify them Pyshp ignores them anyway. When you save files you can specify a base file name that is used for all three file types. Or you can specify a nmae for one or more file types. In that case, any file types not assigned will not save and only file types with file names will be saved. If you do not specify any file names (i.e. save()), then a unique file name is generated with the prefix "shapefile_" followed by random characters which is used for all three files. The unique file name is returned as a string.

```
>>> targetName = w.save()
>>> assert("shapefile_" in targetName)
```

Saving to File-Like Objects

Just as you can read shapefiles from python file-like objects you can also write them.

```
>>> try:
...     from StringIO import StringIO
... except ImportError:
...     from io import BytesIO as StringIO
>>> shp = StringIO()
>>> shx = StringIO()
>>> dbf = StringIO()
>>> w.saveShp(shp)
>>> w.saveShx(shx)
>>> w.saveDbf(dbf)
>>> # Normally you would call the "StringIO.getvalue()" method on these objects.
>>> shp = shx = dbf = None
```

Editing Shapefiles

The Editor class attempts to make changing existing shapefiles easier by handling the reading and writing details behind the scenes.

Let's add shapes to existing shapefiles:

Add a point to a point shapefile

```
>>> e = shapefile.Editor(shapefile="shapefiles/test/point.shp")
>>> e.point(0,0,10,2)
>>> e.record("Appended","Point")
>>> e.save('shapefiles/test/point')
```

Add a new line to a line shapefile:

```
>>> e = shapefile.Editor(shapefile="shapefiles/test/line.shp")
>>> e.line(parts=[[[10,5],[15,5],[15,1],[13,3],[11,1]]])
>>> e.record('Appended','Line')
>>> e.save('shapefiles/test/line')
```

Add a new polygon to a polygon shapefile:

```
>>> e = shapefile.Editor(shapefile="shapefiles/test/polygon.shp")
>>> e.poly(parts=[[[5.1,5],[9.9,5],[9.9,1],[7.5,3],[5.1,1]]])
>>> e.record("Appended","Polygon")
>>> e.save('shapefiles/test/polygon')
```

Remove the first point in each shapefile - for a point shapefile that is the first shape and record"

```
>>> e = shapefile.Editor(shapefile="shapefiles/test/point.shp")
>>> e.delete(0)
>>> e.save('shapefiles/test/point')
```

Remove the last shape in the polygon shapefile.

```
>>> e = shapefile.Editor(shapefile="shapefiles/test/polygon.shp")
>>> e.delete(-1)
>>> e.save('shapefiles/test/polygon')
```

Python __geo_interface__

The Python __geo_interface__ convention provides a data interchange interface among geospatial Python libraries. The interface returns data as GeoJSON. More information on the __geo_interface__ protocol can be found at: https://gist.github.com/sgillies/2217756. More information on GeoJSON is available at http://geojson.org http://geojson.org.

```
>>> s = sf.shape(0)
>>> s.__geo_interface__["type"]
'MultiPolygon'
```