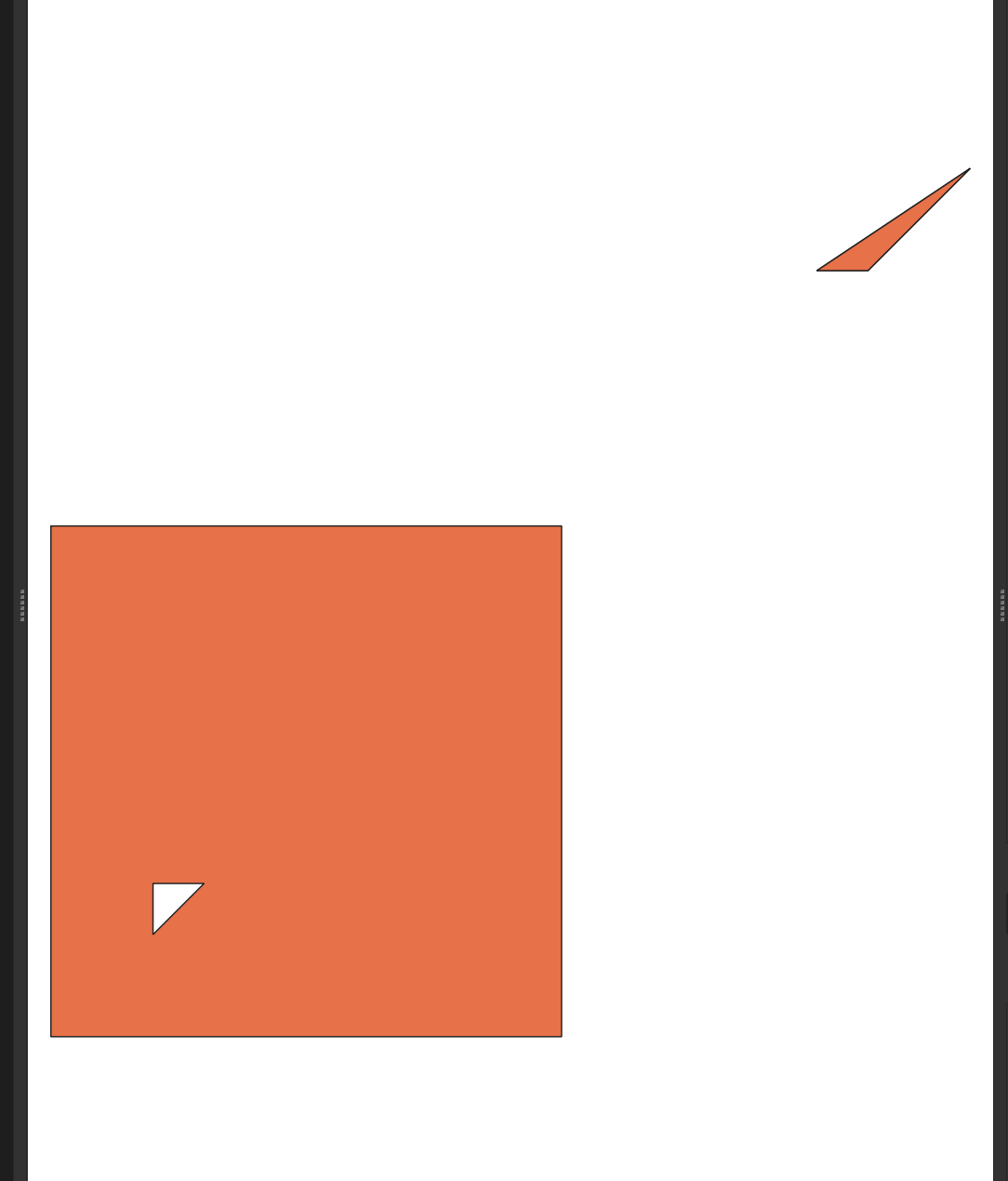
《Python数据处理与分析》

实验报告

|  |  |  |
| --- | --- | --- |
| 学院 | ： | 地球科学学院 |
| 年级专业 | ： | 2021级地理信息科学 |
| 姓 名 | ： | 周全 |
| 学号 | ： | 20211170132 |
| 教师 | ： | 杨克诚 |
| 撰写日期 | ： | 2023.12.4 |

云南大学地球科学学院

最后的结果，

老师这是从我写的博客翻译过来的，原文链接

<https://bre01.github.io/posts/read-shapefile-only-using-python-builtin-library>

使用 python 读取 shapefile——困难的方法

仅使用Python内置的libraray读取shapefile

Shapefile 是开源的，嗯，有点像。 ESRI没有提供任何工具来轻松阅读它， 但至少它提供了技术 规范。 因此，编写我们自己的库来读取二进制并将其转换为某种格式并不困难

在博客中，我将尝试使用python内置的库来读取shapefile，并将其转换为Geojson。 所以我们可以在我们的网络应用程序中使用它。 Shapefile 通常包含三个文件：.shp、.shx、.dbf。 .shp 文件包含几何信息， .shx 文件包含索引信息，.dbf 文件包含属性信息。 可能还有其他文件，例如地图项目的 proj 文件，这些文件不是必需的，通常由 ESRI 专用，因此我们不会尝试读取和转换它。

通过阅读规范，我们可以发现shp文件非常直观，它只包含一种几何类型， 如果几何类型是点，那么记录的是一个点，如果几何类型是折线， 那么记录就是形成几何图形的多个点。

图像

文件头占用100字节（50字） 保留了很多字节，我们需要的是文件长度和几何类型 分别设置在字节24-27和字节32-35。 我们需要注意的一件事是文件长度的字节顺序是大端，而几何类型的字节顺序是小端。 之后有四个双精度数字，表示 shapefile 的边界框

图像

知道了这两个关键信息后，我们就可以开始读取记录了。

每条记录的Header占用8个字节（4个字），我一直强调word因为它在白皮书中被广泛使用，并且记录内容长度是可变的并在记录头中指定。

图像

每个记录头包含记录号（整数）和记录长度（整数），两者都是大端字节序，分别设置在字节0-3和字节4-7。 记录编号以 1 开头并按顺序递增。记录长度是记录内容的长度，以16位字表示。 所以每条记录占用（2字节\*记录长度+8）字节

shapefile 有很多几何类型，但我们只关注三种最常见的类型：点、折线和多边形。 一个点由一对按 X、Y 顺序排列的双精度坐标组成。 point的记录内容很简单，只包含该点的x和y坐标， 两者都是双精度浮点数，并且都是小尾数，

图像

图像

而折线和多边形包含点列表；

图像

它还有一个边界框，用于加速渲染过程。 一条折线可以分为多个部分，每个部分都是一个点列表， 我们可以在字节 44~(x-1) 处找到零件信息，x = 44 + 4 \* num\_parts，零件部分存储每个零件的第一个点的索引 下面的点部分是点数组。

例如，如果部分部分是 [0, 3, 5]，则第一个部分是点部分中的前三个点。

那么多边形呢？ 实际上，我们可以将多边形视为折线的特例， 唯一的区别是多边形的最后一个点与第一个点相同。

有了这些信息，我们就可以开始编写自己的库来读取 shapefile 的几何信息< /span>

首先，我们需要一个类来表示 shapefile。

class Shp(object):

def \_\_init\_\_(self,\*agrs):

#set the path of the first parameter as the path of the shapefile when Shp is initialized

self.path=agrs[0]

self.shp=None

#a list to hold file opened, so we can close them later

self.\_file\_to\_close()=[]

self.shpLength=None

self.shapeType=None

self.numShapes=None

self.shapeName=None

self.encoding = 'utf-8'

self.encoding\_errors = 'strict'

为了读取 .shp 文件，我们需要的第一个信息是 shp 文件的标头， 因此我们加载 shp 文件并在构造函数中读取标头。

class Shp(object):

def \_\_init\_\_(self,\*args)

...

self.load()

def load(self,shapeName):

(shapeName, ext) = os.path.splitext(shapefile)

self.shapeName = shapeName

self.load\_shp(shapeName)

def load\_shp(self,shapeName):

self.shp = open("%s.%s" % (shapefile\_name, "shp"), "rb")

self.shx = open("%s.%s" % (shapefile\_name, "shx"), "rb")

self.\_file\_to\_close.append(self.shp)

self.\_file\_to\_close.append(self.shx)

if (self.shp):

self.load\_shp\_header()

if (self.shx):

self.load\_shx\_header()

def load\_shp\_header(self):

shp = self.shp

shp.seek(24)

# file length big endian

self.shpLength = unpack(">i", shp.read(4))[0] \* 2

# from word(2 bytes) to byte

# shape type :little endian

shp.seek(32)

#small endian, one integer, 4 bytes

self.shapeType = unpack("<i", shp.read(4))[0] #unpack returns a tuple,

#so we need to get the first element of the tuple

#four double that represent the bounding box

self.bbox = \_Array("d", unpack("<4d", shp.read(32)))

#two double that represent the z box

self.zbox = \_Array("d", unpack("<2d", shp.read(16)))

#two double that represent the m box

self.mbox = []

for m in \_Array("d", unpack("<2d", shp.read(16))):

NODATA = -10e38 #according to the whitepaper, number < -10e38 is considered as nodata

#we can also specify the nodata value at the top of the file

if m > NODATA:

self.mbox.append(m)

else:

self.mbox.append(None)

所以shp的头已经被读到内存了 我们可以开始读取每个要素几何的内容 我们可以返回一个生成器，生成每个特征的几何形状， 当我们有一个很大的形状文件时，生成器真的很有帮助

# we create a Shape class to represent the geometry and header of each feature

class Shape(object):

def \_\_init\_\_(

self, oid=None

):

#when we init a Shape, we give it oid

self.shapeType = None

self.points = []

self.parts = []

#we set these public fields so

#that we can set it outside the class

if partTypes:

self.partTypes = partTypes

# and a dict to silently record any errors encountered

self.\_errors = {}

# add oid which is the index of the feature in the shapefile

if oid is not None:

self.\_\_oid = oid

else:

self.\_\_oid = -1

我们定义了一种延迟读取每个特征的几何形状的方法 通过利用发电机， 仅当我们需要时才读取几何图形

class Shp(object):

...

def iterShapes(self, bbox=None):

"""Returns a generator of shapes in a shapefile. Useful

for handling large shapefiles.

To only read shapes within a given spatial region, specify the 'bbox'

arg as a list or tuple of xmin,ymin,xmax,ymax.

"""

shp = self.\_\_getFileObj(self.shp)

# Found shapefiles which report incorrect

# shp file length in the header. Can't trust

# that so we seek to the end of the file

# and figure it out.

shp.seek(0, 2)

shpLength = shp.tell()

shp.seek(100)

# Instead iterate until reach end of file

# Collect the offset indices during iteration

i = 0

offsets = []

pos = shp.tell()

while pos < shpLength:

offsets.append(pos)

#when we create a Shape, we give it oid

shape = self.setShapeIndexAndRead(oid=i, )

#because of the use of generator,

#the shape is read when each \_\_next\_\_ is called

pos = shp.tell()

if shape:

yield shape

i += 1

# Entire shp file consumed

# Update the number of shapes and list of offsets

assert i == len(offsets)

self.numShapes = i

self.\_offsets = offsets

def setShapeIndexAndRead(self, oid=None):

"""Returns the header info and geometry for a single shape."""

f = self.\_\_getFileObj(self.shp)

single\_shape = Shape(oid=oid)

nParts = nPoints = zmin = zmax = mmin = mmax = None

(recIndex, recLength) = unpack(">2i", f.read(8))

#print(recIndex,"||",oid)

#the recIndex =oid+1 because the oid starts from 0

#while recIndex starts from 1

# Determine the start of the next record

next = f.tell() + (2 \* recLength)

shapeType = unpack("<i", f.read(4))[0]

single\_shape.shapeType = shapeType

# For Null shapes create an empty points list for consistency

if shapeType == 0:

single\_shape.points = []

# All shape types capable of having a bounding box

elif shapeType in (3, 5, 8, 13, 15, 18, 23, 25, 28, 31):

single\_shape.bbox = \_Array("d", unpack("<4d", f.read(32)))

# if bbox specified and no overlap, skip this shape

# Shape types with parts

if shapeType in (3, 5, 13, 15, 23, 25, 31):

nParts = unpack("<i", f.read(4))[0]

# Shape types with points

if shapeType in (3, 5, 8, 13, 15, 18, 23, 25, 28, 31):

nPoints = unpack("<i", f.read(4))[0]

# Read parts

if nParts:

single\_shape.parts = \_Array("i", unpack("<%si" % nParts, f.read(nParts \* 4)))

# Read points - produces a list of [x,y] values

if nPoints:

flat = unpack("<%sd" % (2 \* nPoints), f.read(16 \* nPoints))

#make the points into the form of [x,y]

single\_shape.points = list(zip(\*(iter(flat),) \* 2))

# Read a single point

if shapeType in (1, 11, 21):

#only read two point, simply make it a array

single\_shape.points = [\_Array("d", unpack("<2d", f.read(16)))]

# Seek to the end of this record as defined by the record header because

# the shapefile spec doesn't require the actual content to meet the header

# definition. Probably allowed for lazy feature deletion.

f.seek(next)

return single\_shape

现在我们有了每个形状的几何形状 既然我们正在处理这个问题， 折线，多边形，我们只关心每个形状的点和部分

关于 shapefile 的陷阱multi

GeoJson 有一个多重 几何图形的概念，例如 MultiPoint、MultiLineString、MultiPolygon

Shapefile没有这个概念，但它有零件的概念。

多边形和折线都有部分，但这并不意味着它们是多重几何图形

拥有多个部分并不意味着它是一个多重几何图形

转换为GeoJson时，我们要通过计算几何图形来判断是否为multi几何图形

零件可以有一个外环和任意数量的内环， 外环的数量决定了它是否是一个多重几何图形

多重多边形

图像

多边形

图像

零件部分中的索引是每个零件（外部或内部线）的第一个点相对于

如果只有一个部分，那么它当然是一个多边形或折线（在GeoJson的概念中）

如果外环数为1，则它是多边形

如果外环的数量大于 1，则它是 MultiPolygon

零件是外部还是内部取决于零件中点的缠绕顺序

图像

我们可以编写一个函数，它接受环列表并返回多边形列表 这是来自开源库pyshp的一个非常复杂的函数，它使用其他函数，例如 is\_cw、ring\_bbox、bbox\_contains、ring\_sample、ring\_contains\_point、signed\_area 你可以在pyshp的源代码中找到这些函数的实现 你也可以在我的Github上查看简洁的代码

def organize\_polygon\_rings(rings, return\_errors=None):

"""Organize a list of coordinate rings into one or more polygons with holes.

Returns a list of polygons, where each polygon is composed of a single exterior

ring, and one or more interior holes. If a return\_errors dict is provided (optional),

any errors encountered will be added to it.

Rings must be closed, and cannot intersect each other (non-self-intersecting polygon).

Rings are determined as exteriors if they run in clockwise direction, or interior

holes if they run in counter-clockwise direction. This method is used to construct

GeoJSON (multi)polygons from the shapefile polygon shape type, which does not

explicitly store the structure of the polygons beyond exterior/interior ring orientation.

"""

# first iterate rings and classify as exterior or hole

exteriors = []

holes = []

for ring in rings:

# shapefile format defines a polygon as a sequence of rings

# where exterior rings are clockwise, and holes counterclockwise

if is\_cw(ring):

# ring is exterior

exteriors.append(ring)

else:

# ring is a hole

holes.append(ring)

# if only one exterior, then all holes belong to that exterior

if len(exteriors) == 1:

# exit early

poly = [exteriors[0]] + holes

polys = [poly]

return polys

# multiple exteriors, ie multi-polygon, have to group holes with correct exterior

# shapefile format does not specify which holes belong to which exteriors

# so have to do efficient multi-stage checking of hole-to-exterior containment

elif len(exteriors) > 1:

# exit early if no holes

if not holes:

polys = []

for ext in exteriors:

poly = [ext]

polys.append(poly)

return polys

# first determine each hole's candidate exteriors based on simple bbox contains test

hole\_exteriors = dict([(hole\_i, []) for hole\_i in xrange(len(holes))])

exterior\_bboxes = [ring\_bbox(ring) for ring in exteriors]

for hole\_i in hole\_exteriors.keys():

hole\_bbox = ring\_bbox(holes[hole\_i])

for ext\_i, ext\_bbox in enumerate(exterior\_bboxes):

if bbox\_contains(ext\_bbox, hole\_bbox):

hole\_exteriors[hole\_i].append(ext\_i)

# then, for holes with still more than one possible exterior, do more detailed hole-in-ring test

for hole\_i, exterior\_candidates in hole\_exteriors.items():

if len(exterior\_candidates) > 1:

# get hole sample point

ccw = not is\_cw(holes[hole\_i])

hole\_sample = ring\_sample(holes[hole\_i], ccw=ccw)

# collect new exterior candidates

new\_exterior\_candidates = []

for ext\_i in exterior\_candidates:

# check that hole sample point is inside exterior

hole\_in\_exterior = ring\_contains\_point(

exteriors[ext\_i], hole\_sample

)

if hole\_in\_exterior:

new\_exterior\_candidates.append(ext\_i)

# set new exterior candidates

hole\_exteriors[hole\_i] = new\_exterior\_candidates

# if still holes with more than one possible exterior, means we have an exterior hole nested inside another exterior's hole

for hole\_i, exterior\_candidates in hole\_exteriors.items():

if len(exterior\_candidates) > 1:

# exterior candidate with the smallest area is the hole's most immediate parent

ext\_i = sorted(

exterior\_candidates,

key=lambda x: abs(signed\_area(exteriors[x], fast=True)),

)[0]

hole\_exteriors[hole\_i] = [ext\_i]

# separate out holes that are orphaned (not contained by any exterior)

orphan\_holes = []

for hole\_i, exterior\_candidates in list(hole\_exteriors.items()):

if not exterior\_candidates:

orphan\_holes.append(hole\_i)

del hole\_exteriors[hole\_i]

continue

# each hole should now only belong to one exterior, group into exterior-holes polygons

polys = []

for ext\_i, ext in enumerate(exteriors):

poly = [ext]

# find relevant holes

poly\_holes = []

for hole\_i, exterior\_candidates in list(hole\_exteriors.items()):

# hole is relevant if previously matched with this exterior

if exterior\_candidates[0] == ext\_i:

poly\_holes.append(holes[hole\_i])

poly += poly\_holes

polys.append(poly)

# add orphan holes as exteriors

for hole\_i in orphan\_holes:

ext = holes[hole\_i]

# add as single exterior without any holes

poly = [ext]

polys.append(poly)

if orphan\_holes and return\_errors is not None:

return\_errors["polygon\_orphaned\_holes"] = len(orphan\_holes)

return polys

# no exteriors, be nice and assume due to incorrect winding order

else:

if return\_errors is not None:

return\_errors["polygon\_only\_holes"] = len(holes)

exteriors = holes

# add as single exterior without any holes

polys = [[ext] for ext in exteriors]

return polys

从 shapefile 转换而来的 Geojson 将包含一个对象 具有设置为“FeatureCollection”的类型属性。和一个特征对象 由一系列要素组成，每个要素都有一个几何对象和一个属性对象，以及一个类型属性 设置为功能

这是多边形的 json 结构 无尽的 [ 和 ] 会令人沮丧

{

"type": "FeatureCollection",

"features": [

{

"type": "Feature",

"properties":null,

"geometry": {

"type": "Polygon",

"coordinates": [

[[20.0, 20.0], [20.0, 30.0], [30.0, 30.0], [20.0, 20.0]],

[[0.0, 0.0], [100.0, 0.0], [100.0, 100.0], [0.0, 100.0], [0.0, 0.0]]

]

}

},

{

"type": "Feature",

"properties": null,

"geometry":

{

"type": "Polygon",

"coordinates": [

[[150.0, 150.0], [160.0, 150.0], [180.0, 170.0], [150.0, 150.0]]

]

}

},

],

"bbox": [0.0, 0.0, 180.0, 170.0]}

如果每个几何体的坐标都是这样

"coordinates":[1.0,2.0]

这是一个点

"coordinates":[[1.0,2.0],[3.0,4.0]]

LineString 或 MultiPoint 取决于类型属性

"coordinates":[

[[0.0,10.0],[10.0,10.0],[10.0,0.0],[0.0,0.0]]

[[1.0,2.0],[3.0,4.0],[5.0,6.0],[1.0,2.0]],

]

Polygon 或 MultiLineString 取决于类型属性

"coordinates":[

[

[[1.0,2.0],[3.0,4.0],[5.0,6.0],[1.0,2.0]],

[[7.0,8.0],[9.0,10.0],[11.0,12.0],[7.0,8.0]]],

[

[[13.0,14.0],[15.0,16.0],[17.0,18.0],[13.0,14.0]]

]

]

multipolygon 有两个多边形

第一个多边形有两个环，第一个环是外环，第二个环是内环 第二个多边形只有一个外环

Shape 类可以使用 \_\_json\_\_ 方法转换为 json

class Shape(object):

...

def \_\_json\_\_(self):

if self.shapeType=="Point":

# the point only contains one point

return {"type": "Point", "coordinates": tuple(self.points[0])}

elif self.shapeType=="Polyline":

if len(self.parts) == 0:

# the shape has no coordinate information, i.e. is 'empty'

# the geojson spec does not define a proper null-geometry type

# however, it does allow geometry types with 'empty' coordinates to be interpreted as null-geometries

return {"type": "LineString", "coordinates": []}

elif len(self.parts) == 1:

# linestring

return {

"type": "LineString",

"coordinates": [tuple(p) for p in self.points],

}

else:

# multilinestring

ps = None

coordinates = []

for part in self.parts:

if ps == None:

ps = part

continue

else:

coordinates.append([tuple(p) for p in self.points[ps:part]])

ps = part

else:

coordinates.append([tuple(p) for p in self.points[part:]])

return {"type": "MultiLineString", "coordinates": coordinates}

elif self.shapeType=="Polygon":

if len(self.parts) == 0:

# the shape has no coordinate information, i.e. is 'empty'

# the geojson spec does not define a proper null-geometry type

# however, it does allow geometry types with 'empty' coordinates to be interpreted as null-geometries

return {"type": "Polygon", "coordinates": []}

else:

# get all polygon rings

rings = []

for i in range(len(self.parts)):

# get indexes of start and end points of the ring

start = self.parts[i]

try:

end = self.parts[i + 1]

except IndexError:

end = len(self.points)

# extract the points that make up the ring

ring = [tuple(p) for p in self.points[start:end]]

rings.append(ring)

# organize rings into list of polygons, where each polygon is defined as list of rings.

# the first ring is the exterior and any remaining rings are holes (same as GeoJSON).

polys = organize\_polygon\_rings(rings, self.\_errors)

# return as geojson

if len(polys) == 1:

return {"type": "Polygon", "coordinates": polys[0]}

else:

return {"type": "MultiPolygon", "coordinates": polys}

else:

raise Exception("Unknown shape type")

每个 Shape 实例都将转换为字典作为要添加到功能列表中的功能 我们现在还将每个功能的属性设置为 None，

class Shp():

...

def iterShapeRecords(self):

# the number of shape and record should be the same

# in a typical(correct) shapefile

# so we combine dbf attribute and shape into the a shapeRecord

# but for now we only care about shape so set the record to None

# the None will be convert to null in json

for shape in self.iterShapes():

yield ShapeRecord(shape=shape, record=None)

我们设置一个 ShapeRecord 类来保存形状和记录

class ShapeRecord(object):

"""A ShapeRecord object containing a shape along with its attributes.

Provides the \_\_json\_\_ property to return a Feature dictionary."""

def \_\_init\_\_(self, shape=None, record=None):

self.shape = shape

self.record = record

# line 493

@property

def \_\_json\_\_(self):

# we don't have properties now, so return properties as null

return {

"type": "Feature",

"properties": None,

"geometry": self.shape.\_\_json\_\_

}

所以我们可以将 ShapeRecords 包装在 FeatureCollection 中

class Shp():

...

def \_\_json\_\_(self):

"""Returns a GeoJSON FeatureCollection representation of the shapefile."""

return {

"type": "FeatureCollection",

"features": [sr.\_\_json\_\_ for sr in self.iterShapeRecords()],

"bbox": self.bbox,

}

主要功能就像这样 我们可以使用 json 将 dict 转储出来

a=Shp("test.shp")

geoJson=a.\_\_json\_\_()

with open("test.json","w") as f:

json.dump(geoJson,f)