Intro to GDAL

Intro to GDAL

Overview

1. What is GDAL?
2. Caveats and Limitations
3. Basic Data Model
4. Considerations for Reading Raster Data
5. Things to Keep in Mind

Intro to GDAL

Intro to GDAL

- GDAL is the Geospatial Data Abstrction Library
- GDAL includes OGR, the part of the library for vector data
- GDAL includes a number of incredibly useful command line utilities you should also learn
- For the purpose of this class we will define GDAL to be the python bindings of the raster part of the library

Intro to GDAL

- For the most part, GDAL is simply a mechanism to get geospatial raster data into python scripts as numpy arrays
- Some advanced functions are included, such as `RasterizeLayer()`, `Polygonize()`, `ReprojectImage()`, and `CreateAndReprojectImage()`
- Most often your will have to "roll your own" tools, implementing all of the processing required

Caveats and Limitations

Caveats and Limitations

- Same as OGR:
  - Verbose
  - Not pythonic
  - Low level
  - Cryptic error messages

Basic Data Model

Basic Data Model

- GDAL data model is hierarchical like OGR, but with much less nesting
- Looks a little something like this:
  - (like an OGR data source)
    - Coordinate System
    - GeoTransformation / Ground Control Points (GCPs)
    - Subdataset(s) (like a dataset within a dataset)
    -
      - Data
      -
      -

Basic Data Model

- Datasets can be opened similarly to data sources in OGR:

```
from osgeo import gdal

# not really necessary
gdal.AllRegister()

# open the raster file as a read-only dataset object
dataset = gdal.Open(r"C:\data\rasters\elevation.img")

# open the same file as writable
writable = gdal.Open(r"C:\data\rasters\elevation.img", gdal.GA_Update)

# Note: gdal constants are defined in osgeo.gdalconst, so you can
from osgeo.gdalconst import *

# Note also that GA_ReadOnly and GA_Update are simply just 0 and 1

# Again, "close" things by setting to None
writable = None
```

Basic Data Model

- Coordinate systems are simply just OSR spatial references

```
print dataset.GetProjection()

#PROJCS["NAD_1983_HARN_StatePlane_Washington_North_FIPS_4601",GEOGCS["GCS_North_A
#merican_1983_HARN",DATUM["NAD83_High_Accuracy_Reference_Network",SPHEROID["GRS_1
#980",6378137.0,298.257222101]],PRIMEM["Greenwich",0.0],UNIT["Degree",0.017453292
#5199433]],PROJECTION["Lambert_Conformal_Conic_2SP"],PARAMETER["False_Easting",50
#0000.0],PARAMETER["False_Northing",0.0],PARAMETER["Central_Meridian",-120.833333
#3333333],PARAMETER["Standard_Parallel_1",47.5],PARAMETER["Standard_Parallel_2",4
#8.73333333333333],PARAMETER["Latitude_Of_Origin",47.0],UNIT["Meter",1.0]]
```

Basic Data Model

- GeoTransforms describe the relationship of pixels to georeferenced coordinates
    - GCPs are not often used, so for the sake of simplicity we will not use them in this course

```
gt = dataset.GetGeoTransform()
print gt

# (491021.25684745074, 20.0, 0.0, 64365.903204183094, 0.0, -20.0)
```

Basic Data Model

- We see `GetGeoTransform()` return six coefficients to map x and y to col and row:
  - `gt[0]` is the x coord of the raster origin
  - `gt[3]` is the y coord of the raster origin
  - For north-up rasters:
    - `gt[2]` and `gt[4]` are `0`
    - `gt[1]` is the pixel width
    - `gt[5]` is the pixel height
- Geo coords are given by:
  - `x_geo = gt[0] + x_pixel*gt[1] + y_pixel*gt[2]`
  - `y_geo = gt[3] + x_pixel*gt[4] + x_pixel*gt[5]`
- Reverse for pixel coords from geo coords

Basic Data Model

- Subdatasets are supported by a small collection of formats
    - Allow a dataset "container" to hold a number of rasters covering the same geographic area with the same resolution

```
# HDF4, as used by MODIS, uses subdatasets
ds_with_subds = gdal.Open(r"C:\data\rasters\modis_image.hdf")

# returns a list of subdatasets
subds = ds_with_subds.GetSubDatasets()

# to use a subdataset, need to open the path returned by
# GetSubdatasets() using gdal.Open
```

Basic Data Model

- A raster dataset stores the actual data within band objects
- A raster dataset can have multiple bands
    - bands can be different data types
- Bands have a method to `ReadAsArray()`:

```
# get number of bands in a dataset
bandcount = dataset.RasterCount

# read in a band by its index
# NOTE: index starts at 1
band = dataset.GetRasterBand(1)

# ReadAsArray takes four parameters
xoff = 0
yoff = 0
xsize = dataset.RasterXSize
ysize = dataset.RasterYSize

array = band.ReadAsArray(xoff, yoff, xsize, ysize)
# now the data from this band is in a 2-d numpy array
```

Considerations for Reading Raster Data

Considerations for Reading Raster Data

- We just saw how to read the data in a band into an array all at once
    - This is fine, and is fast, but is not often efficient from a memory perspective

Considerations for Reading Raster Data

- What if we read each pixel one at a time?

```
rows = dataset.RasterYSize
cols = dataset.RasterXSize

for row in xrange(rows):
    for col in xrange(cols):
        data = band.ReadAsArray(col, row, 1, 1)
        # do something with the pixel data
```

Considerations for Reading Raster Data

- One pixel at a time is low-memory, but very costly from an efficiency perspective, i.e.,
- Any other ideas?

Considerations for Reading Raster Data

- How about by columns?

```
rows = dataset.RasterYSize
cols = dataset.RasterXSize

for col in xrange(cols):
    data = band.ReadAsArray(col, 0, 1, rows)
    # do something with the column data
```

Considerations for Reading Raster Data

- How about by rows?

```
rows = dataset.RasterYSize
cols = dataset.RasterXSize

for row in xrange(rows):
    data = band.ReadAsArray(0, row, cols, 1)
    # do something with the row data
```

Considerations for Reading Raster Data

- Raster data is typically stored as blocks within the file structure
    - Block can be tiles in a tiled raster format, or rows in a non-tiled format
- To combine processing and memory efficiency, read via the blocks in the file

Considerations for Reading Raster Data

- We will start by reading a row of blocks

```
# arbitrary block size to illustrate concept
# first number is columns in block
# second number is rows in block
blocksize = [3, 3]

rows = dataset.RasterYSize
cols = dataset.RasterXSize

for row in xrange(0, rows, blocksize[1]):
    if row + blocksize < rows:
        row_read_size = blocksize[1]
    else:
        row_read_size = rows - row

    data = band.ReadAsArray(0, row, cols, row_read_size)
    # do something with the block row data here
```

Considerations for Reading Raster Data

- Now how about reading via blocks from the image

```
# get the blocksize from the band
blocksize = band.GetBlockSize()  # returns a list

rows = dataset.RasterYSize
cols = dataset.RasterXSize

# iterate through rows of blocks
for row in xrange(0, rows, blocksize[1]):
    if row + blocksize[1] < rows:
        row_read_size = blocksize[1]
    else:
        row_read_size = rows - row

    # iterate through columns of blocks
    for col in xrange(0, cols, blocksize[0]):
        if col + blocksize[0] < cols:
            col_read_size = blocksize[0]
        else:
            col_read_size = cols - col

        data = band.ReadAsArray(col, row, col_read_size, row_read_size)
        # do something with the block data here
```

Things to Keep in Mind

Things to Keep in Mind

- Again:
  - Use patterns! Look for examples and emulate what they do to fit your application.
    - GDAL Python Cookbook
    - Geoprocessing with python using Open Source GIS
  - Read the python Gotchas
  - Many examples do not follow the advice of the Gotchas, so read it and internalize what is there so if you see bad examples, you can fix them when you use them in your own code