3
Python Libraries for Geospatial Development

This chapter examines a number of libraries and other tools which can be used for geospatial development in Python.

More specifically, we will cover:

- Python libraries for reading and writing geospatial data
- Python libraries for dealing with map projections
- Libraries for analyzing and manipulating geospatial data directly within your Python programs
- Tools for visualizing geospatial data

Note that there are two types of geospatial tools which are not discussed in this chapter: geospatial databases and geospatial web toolkits. Both of these will be examined in detail later in this book.

Reading and writing geospatial data

While you could in theory write your own parser to read a particular geospatial data format, it is much easier to use an existing Python library to do this. We will look at two popular libraries for reading and writing geospatial data: GDAL and OGR.

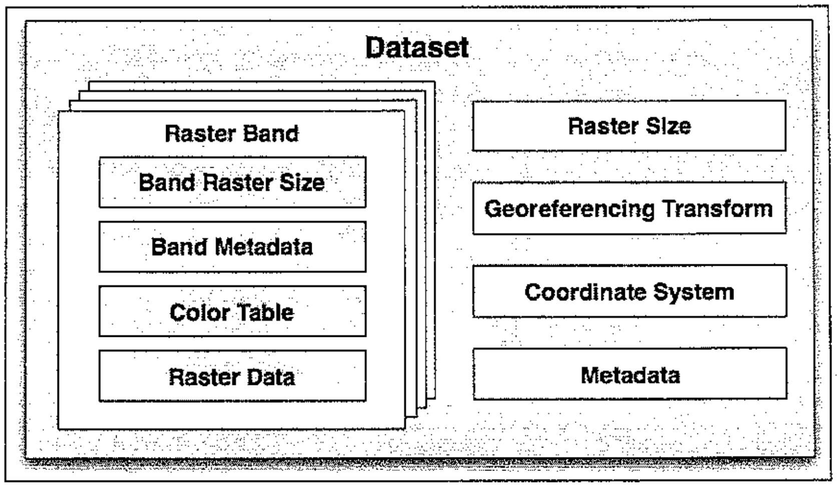
GDAL/OGR

Unfortunately, the naming of these two libraries is rather confusing. **Geospatial Data Abstraction Library** (GDAL), was originally just a library for working with raster geospatial data, while the separate OGR library was intended to work with vector data. However, the two libraries are now partially merged, and are generally downloaded and installed together under the combined name of "GDAL". To avoid confusion, we will call this combined library **GDAL/OGR** and use "GDAL" to refer to just the raster translation library.

A default installation of GDAL supports reading 116 different raster file formats, and writing to 58 different formats. OGR by default supports reading 56 different vector file formats, and writing to 30 formats. This makes GDAL/OGR one of the most powerful geospatial data translators available, and certainly the most useful freely-available library for reading and writing geospatial data.

GDAL design

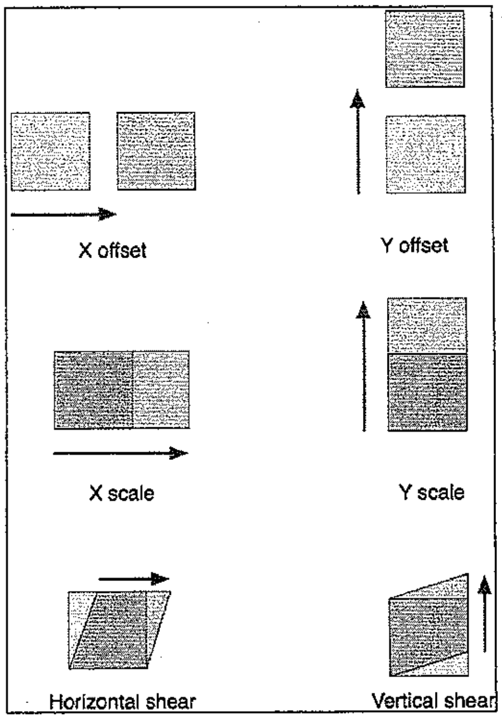
GDAL uses the following data model for describing raster geospatial data:



Let's take a look at the various parts of this model:

- A **dataset** holds all the raster data, in the form of a collection of raster "bands", along with information that is common to all these bands. A dataset normally represents the contents of a single file.

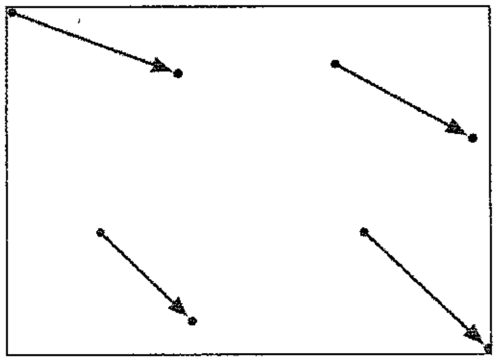
- A **raster band** represents a band, channel, or layer within the image. For example, RGB image data would normally have separate bands for the red, green, and blue components of the image.
- The **raster size** specifies the overall width and height of the image, in pixels.
- The **georeferencing transform** converts from (x, y) raster coordinates into georeferenced coordinates—that is, coordinates on the surface of the earth. There are two types of georeferencing transforms supported by GDAL: affine transformations and ground control points.
    - An **affine transformation** is a mathematical formula allowing the following operations to be applied to the raster data:



More than one of these operations can be applied at once; this allows you to perform sophisticated transforms such as rotations.

[Affine transformations are sometimes referred to as *linear transformations*.]

- **Ground Control Points** (GCPs) relate one or more positions within the raster to their equivalent georeferenced coordinates, as shown in the following figure:



Note that GDAL does not translate coordinates using GCPs—that is left up to the application, and generally involves complex mathematical functions to perform the transformation.

- The **coordinate system** describes the georeferenced coordinates produced by the georeferencing transform. The coordinate system includes the projection and datum, as well as the units and scale used by the raster data.
- The **metadata** contains additional information about the dataset as a whole.

Each raster band contains the following (among other things):

- The **band raster size**: This is the size (number of pixels across and number of lines high) for the data within the band. This may be the same as the raster size for the overall dataset, in which case the dataset is at full resolution, or the band's data may need to be scaled to match the dataset.
- Some **band metadata** providing extra information specific to this band.
- A **color table** describing how pixel values are translated into colors.
- The **raster data** itself.

GDAL provides a number of **drivers** which allow you to read (and sometimes write) various types of raster geospatial data. When reading a file, GDAL selects a suitable driver automatically based on the type of data; when writing, you first select the driver and then tell the driver to create the new dataset you want to write to.

GDAL example code

A **Digital Elevation Model (DEM)** file contains height values. In the following example program, we use GDAL to calculate the average of the height values contained in a sample DEM file. In this case, we use a DEM file downloaded from the GLOBE elevation dataset:

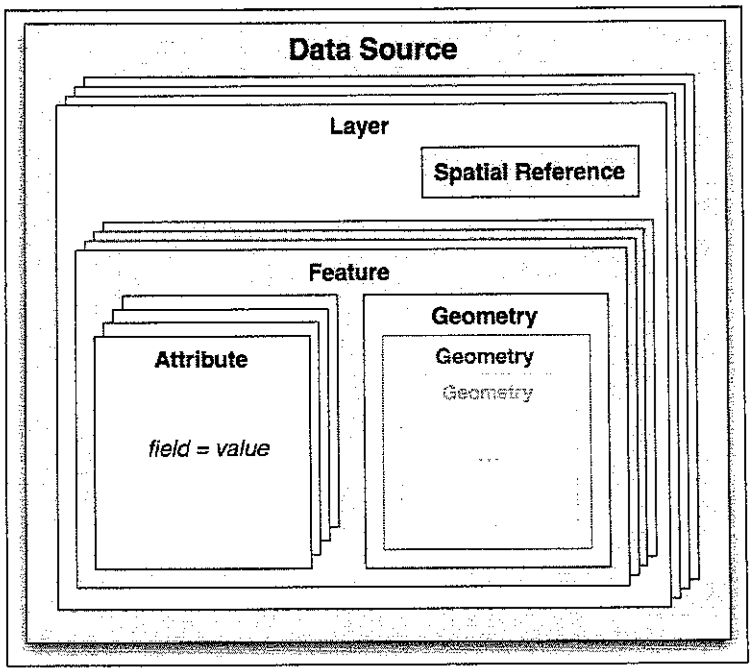
```
from osgeo import gdal,gdalconst
import struct

dataset = gdal.Open("data/e10g")
band = dataset.GetRasterBand(1)

fmt = "<" + ("h" * band.XSize)

totHeight = 0

for y in range(band.YSize):
    scanline = band.ReadRaster(0, y, band.XSize, 1,
                                band.XSize, 1,
                                band.DataType)
    values = struct.unpack(fmt, scanline)

    for value in values:
        if value == -500:
            # Special height value for the sea -> ignore.
            continue

        totHeight = totHeight + value

average = totHeight / (band.XSize * band.YSize)
print "Average height =", average
```

[Please refer to *Chapter 4, Sources of Geospatial Data*, for more information on the GLOBE dataset and how to download the data used in this example.]

As you can see, this program obtains the single raster band from the DEM file, and then reads through it one scanline at a time. We then use the `struct` standard Python library module to read the individual height values out of the scanline. Because the GLOBE dataset uses a special height value of -500 to represent the ocean, we exclude these values from our calculations. Finally, we use the remaining height values to calculate the average height, in meters, over the entire DEM data file.

OGR design

OGR uses the following model for working with vector-based geospatial data:



Let's take a look at this design in more detail:

- The **data source** represents the file you are working with—though it doesn't have to be a file. It could just as easily be a URL or some other source of data.
- The data source has one or more **layers**, representing sets of related data. For example, a single data source representing a country may contain a "terrain" layer, a "contour lines" layer, a "roads" later, and a "city boundaries" layer. Other data sources may consist of just one layer. Each layer has a spatial reference and a list of features.
- The **spatial reference** specifies the projection and datum used by the layer's data.
- A **feature** corresponds to some significant element within the layer. For example, a feature might represent a state, a city, a road, an island, and so on. Each feature has a list of attributes and a geometry.

- The **attributes** provide additional meta-information about the feature. For example, an attribute might provide the name for a city's feature, its population, or the feature's unique ID used to retrieve additional information about the feature from an external database.
- Finally, the **geometry** describes the physical shape or location of the feature. Geometries are recursive data structures that can themselves contain sub-geometries—for example, a "country" feature might consist of a geometry that encompasses several islands, each represented by a subgeometry within the main "country" geometry.

  The geometry design within OGR is based on the Open Geospatial Consortium's "Simple Features" model for representing geospatial geometries. For more information, see `http://www.opengeospatial.org/standards/sfa`.

Like GDAL, OGR also provides a number of drivers which allow you to read (and sometimes write) various types of vector-based geospatial data. When reading a file, OGR selects a suitable driver automatically; when writing, you first select the driver and then tell the driver to create the new data source to write to.

OGR example code

The following example program uses OGR to read through the contents of a shapefile, printing out the value of the `NAME` attribute for each feature along with the geometry type:

```
from osgeo import ogr

shapefile = ogr.Open("TM_WORLD_BORDERS-0.3.shp")
layer = shapefile.GetLayer(0)

for i in range(layer.GetFeatureCount()):
    feature = layer.GetFeature(i)
    name = feature.GetField("NAME")
    geometry = feature.GetGeometryRef()
    print i, name, geometry.GetGeometryName()
```

Documentation

GDAL and OGR are well documented, but with a catch for Python programmers. The GDAL/OGR library and associated command-line tools are all written in C and C++. Bindings are available which allow access from a variety of other languages, including Python, but the documentation is all written for the C++ version of the libraries. This can make reading the documentation rather challenging—not only are all the method signatures written in C++, but the Python bindings have changed many of the method and class names to make them more "pythonic".

Fortunately, the Python libraries are largely self-documenting, thanks to all the docstrings embedded in the Python bindings themselves. This means you can explore the documentation using tools such as Python's built-in `pydoc` utility, which can be run from the command line like this:

```
% pydoc -g osgeo
```

This will open up a GUI window allowing you to read the documentation using a web browser. Alternatively, if you want to find out about a single method or class, you can use Python's built-in `help()` command from the Python command line, like this:

```
>>> import osgeo.ogr
>>> help(osgeo.ogr.DataSource.CopyLayer)
```

Not all the methods are documented, so you may need to refer to the C++ docs on the GDAL website for more information, and some of the docstrings are copied directly from the C++ documentation—but in general the documentation for GDAL/OGR is excellent, and should allow you to quickly come up to speed using this library.

Availability

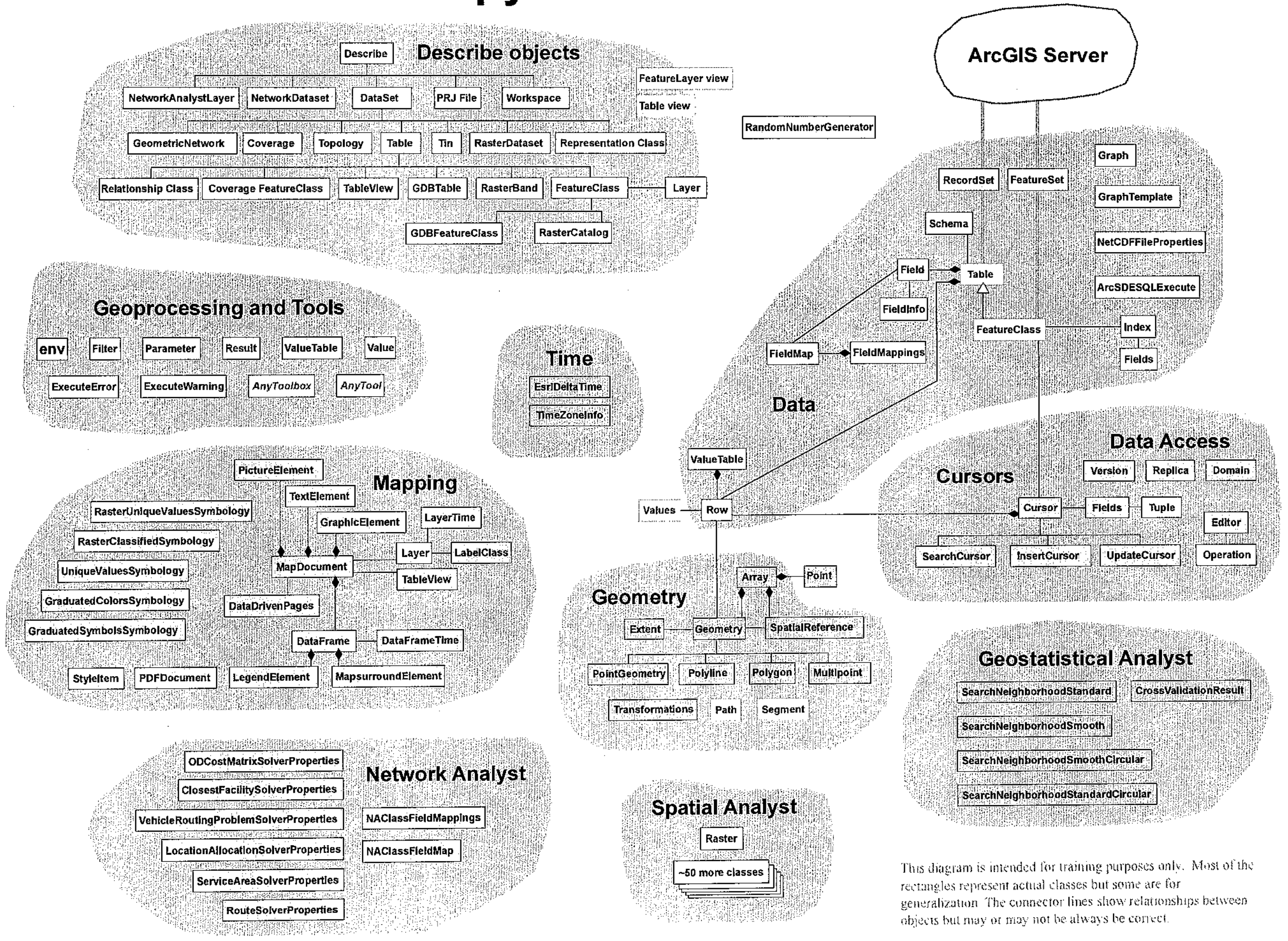
GDAL/OGR runs on modern Unix machines, including Linux and Mac OS X, as well as most versions of Microsoft Windows. The main website for GDAL can be found at:

`http://gdal.org`

The main website for OGR is at:

`http://gdal.org/ogr`

arcpy Classes at 10.1



This diagram is intended for training purposes only. Most of the rectangles represent actual classes but some are for generalization. The connector lines show relationships between objects but may or may not be always be correct.