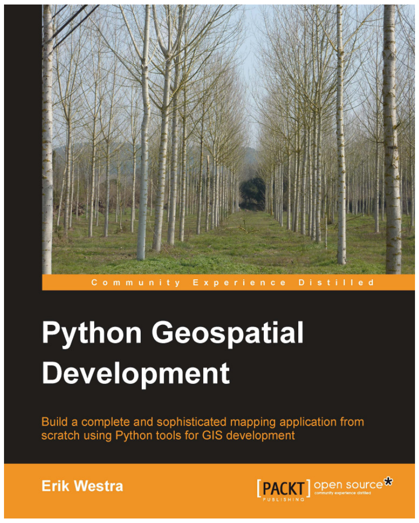
Python Geospatial Development

Erik Westra


Community Experience Distilled
Python Geospatial Development
Build a complete and sophisticated mapping application from scratch using Python tools for GIS development
Erik Westra
PACKT open source

Chapter No. 5
"Working with Geo-Spatial Data in Python"

In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.5 "Working with Geo-Spatial Data in Python"

A synopsis of the book’s content

Information on where to buy this book

About the Author

Erik Westra has been a professional software developer for over 25 years, and has worked almost exclusively in Python for the past decade. Erik's early interest in graphical user-interface design led to the development of one of the most advanced urgent courier dispatch systems used by messenger and courier companies worldwide. In recent years, Erik has been involved in the design and implementation of systems matching seekers and providers of goods and services across a range of geographical areas. This work has included the creation of real-time geocoders and map-based views of constantly changing data. Erik is based in New Zealand, and works for companies worldwide.

"For Ruth,

The love of my life."

Python Geospatial Development

Open Source GIS (Geographic Information Systems) is a growing area with the explosion of Google Maps-based websites and spatially-aware devices and applications. The GIS market is growing rapidly, and as a Python developer you can't afford to be left behind. In today's location-aware world, all commercial Python developers can benefit from an understanding of GIS concepts and development techniques.

Working with geo-spatial data can get complicated because you are dealing with mathematical models of the Earth's surface. Since Python is a powerful programming language with high-level toolkits, it is well-suited to GIS development. This book will familiarize you with the Python tools required for geo-spatial development. It introduces GIS at the basic level with a clear, detailed walkthrough of the key GIS concepts such as location, distance, units, projections, datums, and GIS data formats. We then examine a number of Python libraries and combine these with geo-spatial data to accomplish a variety of tasks. The book provides an in-depth look at the concept of storing spatial data in a database and how you can use spatial databases as tools to solve a variety of geo-spatial problems.

It goes into the details of generating maps using the Mapnik map-rendering toolkit, and helps you to build a sophisticated web-based geo-spatial map editing application using GeoDjango, Mapnik, and PostGIS. By the end of the book, you will be able to integrate spatial features into your applications and build a complete mapping application from scratch.

This book is a hands-on tutorial, teaching you how to access, manipulate, and display geo-spatial data efficiently using a range of Python tools for GIS development.

What This Book Covers

Chapter 1, Geo-Spatial Development Using Python, introduces the Python programming language and the main concepts behind geo-spatial development

Chapter 2, GIS, discusses many of the core concepts that underlie GIS development. It examines the common GIS data formats, and gets our hands dirty exploring U.S. state maps downloaded from the U.S. Census Bureau website

Chapter 3, Python Libraries for Geo-Spatial Development, looks at a number of important libraries for developing geo-spatial applications using Python

Chapter 4, Sources of Geo-Spatial Data, covers a number of sources of freely-available geo-spatial data. It helps you to obtain map data, images, elevations, and place names for use in your geo-spatial applications

Chapter 5, Working with Geo-Spatial Data in Python, deals with various techniques for using OGR, GDAL, Shapely, and pyproj within Python programs to solve real-world problems

Chapter 6, GIS in the Database, takes an in-depth look at the concept of storing spatial data in a database, and examines three of the principal open source spatial databases

Chapter 7, Working with Spatial Data, guides us to implement, test, and make improvements to a simple web-based application named DISTAL. This application displays shorelines, towns, and lakes within a given radius of a starting point. We will use this application as the impetus for exploring a number of important concepts within geo-spatial application development

Chapter 8, Using Python and Mapnik to Generate Maps, helps us to explore the Mapnik map-generation toolkit in depth

Chapter 9, Web Frameworks for Python Geo-Spatial Development, discusses the geo-spatial web development landscape, examining the major concepts behind geo-spatial web application development, some of the main open protocols used by geo-spatial web applications, and a number of Python-based tools for implementing geo-spatial applications that run over the Internet

Chapter 10, Putting it all Together: a Complete Mapping Application, along with the final two chapters, brings together all the topics discussed in previous chapters to implement a sophisticated web-based mapping application called ShapeEditor

Chapter 11, ShapeEditor: Implementing List View, Import, and Export, continues with implementation of the ShapeEditor by adding a "list" view showing the imported Shapefiles, along with the ability to import and export Shapefiles

Chapter 12, ShapeEditor: Selecting and Editing Features, adds map-based editing and feature selection capabilities, completing the implementation of the ShapeEditor application

5
Working with Geo-Spatial Data in Python

In this chapter, we combine the Python libraries and geo-spatial data covered earlier to accomplish a variety of tasks. These tasks have been chosen to demonstrate various techniques for working with geo-spatial data in your Python programs; while in some cases there are quicker and easier ways to achieve these results (for example, using command-line utilities), we will create these solutions in Python so you can learn how to work with geo-spatial data in your own Python programs.

This chapter will cover:

- Reading and writing geo-spatial data in both vector and raster format
- Changing the datums and projections used by geo-spatial data
- Representing and storing geo-spatial data within your Python programs
- Using Shapely to work with points, lines, and polygons
- Converting and standardizing units of geometry and distance

This chapter is formatted like a cookbook, detailing various real-world tasks you might want to perform and providing "recipes" for accomplishing them.

Prerequisites

If you want to follow through the examples in this chapter, make sure you have the following Python libraries installed on your computer:

- GDAL/OGR version 1.7 or later (`http://gdal.org`)
- `pyproj` version 1.8.6 or later (`http://code.google.com/p/pyproj`)
- Shapely version 1.2 or later (`http://trac.gispython.org/lab/wiki/Shapely`)

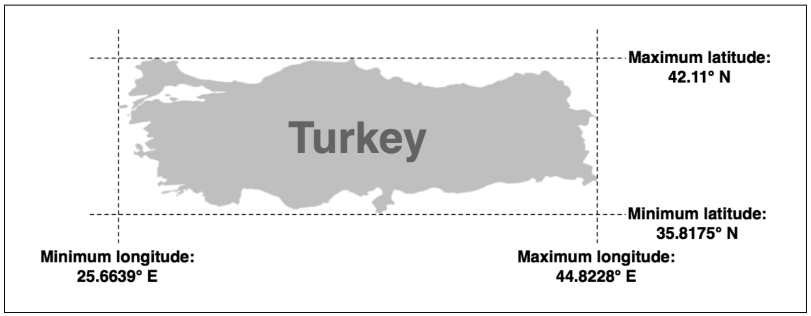
For more information about these libraries and how to use them, including references to the API documentation for each library, please refer to *Chapter 3*.

Reading and writing geo-spatial data

In this section, we will look at some examples of tasks you might want to perform that involve reading and writing geo-spatial data in both vector and raster format.

Task: Calculate the bounding box for each country in the world

In this slightly contrived example, we will make use of a Shapefile to calculate the minimum and maximum latitude/longitude values for each country in the world. This "bounding box" can be used, among other things, to generate a map of a particular country. For example, the bounding box for Turkey would look like this:


Turkey
Maximum latitude: 42.11° N
Minimum latitude: 35.8175° N
Minimum longitude: 25.6639° E
Maximum longitude: 44.8228° E

Start by downloading the World Borders Dataset from:

`http://thematicmapping.org/downloads/world_borders.php`

Decompress the `.zip` archive and place the various files that make up the Shapefile (the `.dbf`, `.prj`, `.shp`, and `.shx` files) together in a suitable directory.

We next need to create a Python program that can read the borders of each country. Fortunately, using OGR to read through the contents of a Shapefile is trivial:

```
import osgeo.ogr

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

for i in range(layer.GetFeatureCount()):
    feature = layer.GetFeature(i)
```

The feature consists of a **geometry** and a set of **fields**. For this data, the geometry is a polygon that defines the outline of the country, while the fields contain various pieces of information about the country. According to the `Readme.txt` file, the fields in this Shapefile include the ISO-3166 three-letter code for the country (in a field named `ISO3`) as well as the name for the country (in a field named `NAME`). This allows us to obtain the country code and name like this:

```
countryCode = feature.GetField("ISO3")
countryName = feature.GetField("NAME")
```

We can also obtain the country's border polygon using:

```
geometry = feature.GetGeometryRef()
```

There are all sorts of things we can do with this geometry, but in this case we want to obtain the bounding box or **envelope** for the polygon:

```
minLong,maxLong,minLat,maxLat = geometry.GetEnvelope()
```

Let's put all this together into a complete working program:

```
# calcBoundingBoxes.py

import osgeo.ogr

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

countries = [] # List of (code,name,minLat,maxLat,
               # minLong,maxLong) tuples.

for i in range(layer.GetFeatureCount()):
    feature = layer.GetFeature(i)
    countryCode = feature.GetField("ISO3")
    countryName = feature.GetField("NAME")
    geometry = feature.GetGeometryRef()
    minLong,maxLong,minLat,maxLat = geometry.GetEnvelope()

    countries.append((countryName, countryCode,
                      minLat, maxLat, minLong, maxLong))

countries.sort()
for name,code,minLat,maxLat,minLong,maxLong in countries:
    print "%s (%s) lat=%0.4f..%0.4f, long=%0.4f..%0.4f" \
% (name, code,minLat, maxLat,minLong, maxLong)
```

Running this program produces the following output:

```
% python calcBoundingBoxes.py
Afghanistan (AFG) lat=29.4061..38.4721, long=60.5042..74.9157
Albania (ALB) lat=39.6447..42.6619, long=19.2825..21.0542
Algeria (DZA) lat=18.9764..37.0914, long=-8.6672..11.9865
...
```

Task: Save the country bounding boxes into a Shapefile

While the previous example simply printed out the latitude and longitude values, it might be more useful to draw the bounding boxes onto a map. To do this, we have to convert the bounding boxes into polygons, and save these polygons into a Shapefile.

Creating a Shapefile involves the following steps:

1. Define the **spatial reference** used by the Shapefile's data. In this case, we'll use the WGS84 datum and unprojected geographic coordinates (that is, latitude and longitude values). This is how you would define this spatial reference using OGR:

   ```
   import osgeo.osr

   spatialReference = osgeo.osr.SpatialReference()
   spatialReference.SetWellKnownGeogCS('WGS84')
   ```

   We can now create the Shapefile itself using this spatial reference:

   ```
   import osgeo.ogr

   driver = osgeo.ogr.GetDriverByName("ESRI Shapefile")
   dstFile = driver.CreateDataSource("boundingBoxes.shp"))
   dstLayer = dstFile.CreateLayer("layer", spatialReference)
   ```

2. After creating the Shapefile, you next define the various fields that will hold the metadata for each feature. In this case, let's add two fields to store the country name and its ISO-3166 code:

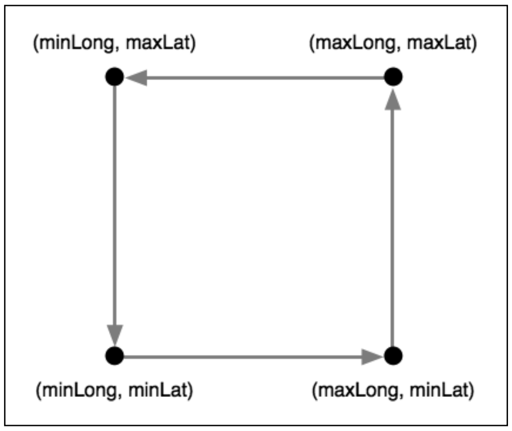
   ```
   fieldDef = osgeo.ogr.FieldDefn("COUNTRY", osgeo.ogr.OFTString)
   fieldDef.SetWidth(50)
   dstLayer.CreateField(fieldDef)

   fieldDef = osgeo.ogr.FieldDefn("CODE", osgeo.ogr.OFTString)
   fieldDef.SetWidth(3)
   dstLayer.CreateField(fieldDef)
   ```

3. We now need to create the geometry for each feature—in this case, a polygon defining the country's bounding box. A polygon consists of one or more **linear rings**; the first linear ring defines the exterior of the polygon, while additional rings define "holes" inside the polygon. In this case, we want a simple polygon with a square exterior and no holes:

   ```
   linearRing = osgeo.ogr.Geometry(osgeo.ogr.wkbLinearRing)
   linearRing.AddPoint(minLong, minLat)
   linearRing.AddPoint(maxLong, minLat)
   linearRing.AddPoint(maxLong, maxLat)
   ```

```
linearRing.AddPoint(minLong, maxLat)
linearRing.AddPoint(minLong, minLat)

polygon = osgeo.ogr.Geometry(osgeo.ogr.wkbPolygon)
polygon.AddGeometry(linearRing)
```

You may have noticed that the coordinate `(minLong, minLat)` was added to the linear ring twice. This is because we are defining line segments rather than just points—the first call to `AddPoint()` defines the starting point, and each subsequent call to `AddPoint()` adds a new line segment to the linear ring. In this case, we start in the lower-left corner and move counter-clockwise around the bounding box until we reach the lower-left corner again:



Once we have the polygon, we can use it to create a feature:

```
feature = osgeo.ogr.Feature(dstLayer.GetLayerDefn())
feature.SetGeometry(polygon)
feature.SetField("COUNTRY", countryName)
feature.SetField("CODE", countryCode)
dstLayer.CreateFeature(feature)
feature.Destroy()
```

Notice how we use the `setField()` method to store the feature's metadata. We also have to call the `Destroy()` method to close the feature once we have finished with it; this ensures that the feature is saved into the Shapefile.

4. Finally, we call the `Destroy()` method to close the output Shapefile:

```
dstFile.Destroy()
```

5. Putting all this together, and combining it with the code from the previous recipe to calculate the bounding boxes for each country in the World Borders Dataset Shapefile, we end up with the following complete program:

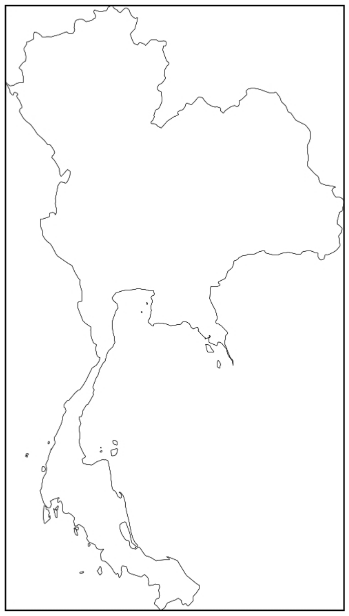
```
# boundingBoxesToShapefile.py

import os, os.path, shutil

import osgeo.ogr
import osgeo.osr

# Open the source shapefile.

srcFile = osgeo.ogr.Open("TM_WORLD_BORDERS-0.3.shp")
srcLayer = srcFile.GetLayer(0)

# Open the output shapefile.

if os.path.exists("bounding-boxes"):
    shutil.rmtree("bounding-boxes")
os.mkdir("bounding-boxes")

spatialReference = osgeo.osr.SpatialReference()
spatialReference.SetWellKnownGeogCS('WGS84')

driver = osgeo.ogr.GetDriverByName("ESRI Shapefile")
dstPath = os.path.join("bounding-boxes", "boundingBoxes.shp")
dstFile = driver.CreateDataSource(dstPath)
dstLayer = dstFile.CreateLayer("layer", spatialReference)

fieldDef = osgeo.ogr.FieldDefn("COUNTRY", osgeo.ogr.OFTString)
fieldDef.SetWidth(50)
dstLayer.CreateField(fieldDef)

fieldDef = osgeo.ogr.FieldDefn("CODE", osgeo.ogr.OFTString)
fieldDef.SetWidth(3)
dstLayer.CreateField(fieldDef)

# Read the country features from the source shapefile.

for i in range(srcLayer.GetFeatureCount()):
    feature = srcLayer.GetFeature(i)
    countryCode = feature.GetField("ISO3")
    countryName = feature.GetField("NAME")
    geometry = feature.GetGeometryRef()
    minLong,maxLong,minLat,maxLat = geometry.GetEnvelope()

    # Save the bounding box as a feature in the output
    # shapefile.
```

```
        linearRing = osgeo.ogr.Geometry(osgeo.ogr.wkbLinearRing)
        linearRing.AddPoint(minLong, minLat)
        linearRing.AddPoint(maxLong, minLat)
        linearRing.AddPoint(maxLong, maxLat)
        linearRing.AddPoint(minLong, maxLat)
        linearRing.AddPoint(minLong, minLat)

        polygon = osgeo.ogr.Geometry(osgeo.ogr.wkbPolygon)
        polygon.AddGeometry(linearRing)

        feature = osgeo.ogr.Feature(dstLayer.GetLayerDefn())
        feature.SetGeometry(polygon)
        feature.SetField("COUNTRY", countryName)
        feature.SetField("CODE", countryCode)
        dstLayer.CreateFeature(feature)
        feature.Destroy()

    # All done.

    srcFile.Destroy()
    dstFile.Destroy()
```

The only unexpected twist in this program is the use of a sub-directory called `bounding-boxes` to store the output Shapefile. Because a Shapefile is actually made up of multiple files on disk (a `.dbf` file, a `.prj` file, a `.shp` file, and a `.shx` file), it is easier to place these together in a sub-directory. We use the Python Standard Library module `shutil` to delete the previous contents of this directory, and then `os.mkdir()` to create it again.

> If you aren't storing the `TM_WORLD_BORDERS-0.3.shp` Shapefile in the same directory as the script itself, you will need to add the directory where the Shapefile is stored to your `osgeo.ogr.Open()` call. You can also store the `boundingBoxes.shp` Shapefile in a different directory if you prefer, by changing the path where this Shapefile is created.

Running this program creates the bounding box Shapefile, which we can then draw onto a map. For example, here is the outline of Thailand along with a bounding box taken from the `boundingBoxes.shp` Shapefile:



We will be looking at how to draw maps in *Chapter 8*.

Task: Analyze height data using a digital elevation map

A DEM (Digital Elevation Map) is a type of raster format geo-spatial data where each pixel value represents the height of a point on the Earth's surface. We encountered DEM files in the previous chapter, where we saw two examples of datasources which supply this type of information: the National Elevation Dataset covering the United States, and GLOBE which provides DEM files covering the entire Earth.

Because a DEM file contains height data, it can be interesting to analyze the height values for a given area. For example, we could draw a histogram showing how much of a country's area is at a certain elevation. Let's take some DEM data from the GLOBE dataset, and calculate a height histogram using that data.

To keep things simple, we will choose a small country surrounded by the ocean: New Zealand.

We're using a small country so that we don't have too much data to work with, and we're using a country surrounded by ocean so that we can check all the points within a bounding box rather than having to use a polygon to exclude points outside of the country's boundaries.

To download the DEM data, go to the GLOBE website (`http://www.ngdc.noaa.gov/mgg/topo/globe.html`) and click on the **Get Data Online** hyperlink. We're going to use the data already calculated for this area of the world, so click on the **Any or all 16 "tiles"** hyperlink. New Zealand is in tile **L**, so click on this tile to download it.

The file you download will be called `l10g.gz`. If you decompress it, you will end up with a file `l10g` containing the raw elevation data.

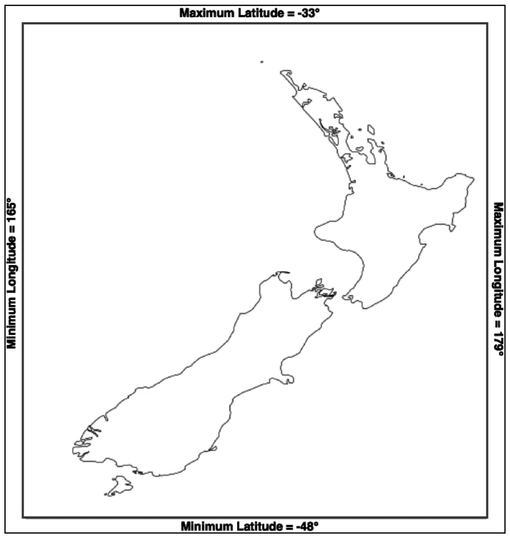
By itself, this file isn't very useful—it needs to be *georeferenced* onto the Earth's surface so that you can match up a height value with its position on the Earth. To do this, you need to download the associated header file. Unfortunately, the GLOBE website makes this rather difficult; the header files for the premade tiles can be found at:

`http://www.ngdc.noaa.gov/mgg/topo/elev/esri/hdr`

Download the file named `l10g.hdr` and place it into the same directory as the `l10g` file you downloaded earlier. You can then read the DEM file using GDAL:

```
import osgeo.gdal
dataset = osgeo.gdal.Open("l10g")
```

As you no doubt noticed when you downloaded the `l10g` tile, this covers much more than just New Zealand—all of Australia is included, as well as Malaysia, Papua New Guinea, and several other east-Asian countries. To work with the height data for just New Zealand, we have to be able to identify the relevant portion of the raster DEM—that is, the range of x,y coordinates which cover New Zealand. We start by looking at a map and identifying the minimum and maximum latitude/longitude values which enclose all of New Zealand, but no other country:



Rounded to the nearest whole degree, we get a long/lat bounding box of (165, -48)…(179, -33). This is the area we want to scan to cover all of New Zealand.

There is, however, a problem—the raster data consists of pixels or "cells" identified by (x,y) coordinates, not longitude and latitude values. We have to convert from longitudes and latitudes into x and y coordinates. To do this, we need to make use of the raster DEM's **affine transformation**.

If you can remember back to *Chapter 3*, an affine transformation is a set of six numbers that define how geographic coordinates (latitude and longitude values) are translated into raster (x,y) coordinates. This is done using two formulas:

```
longitude = t[0] + x*t[1] + y*t[2]
```

```
latitude = t[3] + x*t[4] + y*t[5]
```

Fortunately, we don't have to deal with these formulas directly as GDAL will do it for us. We start by obtaining our dataset's affine transformation:

```
    t = dataset.GetGeoTransform()
```

Using this transformation, we could convert an x,y coordinate into its associated latitude and longitude value. In this case, however, we want to do the opposite—we want to take a latitude and longitude, and calculate the associated x,y coordinate.

To do this, we have to *invert* the affine transformation. Once again, GDAL will do this for us:

```
success,tInverse = gdal.InvGeoTransform(t)
if not success:
    print "Failed!"
    sys.exit(1)
```

[There are some cases where an affine transformation can't be inverted. This is why `gdal.InvGeoTransform()` returns a `success` flag as well as the inverted transformation. With this DEM data, however, the affine transformation should always be invertible.]

Now that we have the inverse affine transformation, it is possible to convert from a latitude and longitude into an x,y coordinate by using:

```
x,y = gdal.ApplyGeoTransform(tInverse, longitude, latitude)
```

Using this, it's easy to identify the minimum and maximum x,y coordinates that cover the area we are interested in:

```
x1,y1 = gdal.ApplyGeoTransform(tInverse, minLong, minLat)
x2,y2 = gdal.ApplyGeoTransform(tInverse, maxLong, maxLat)

minX = int(min(x1, x2))
maxX = int(max(x1, x2))
minY = int(min(y1, y2))
maxY = int(max(y1, y2))
```

Now that we know the x,y coordinates for the portion of the DEM that we're interested in, we can use GDAL to read in the individual height values. We start by obtaining the raster band that contains the DEM data:

```
band = dataset.GetRasterBand(1)
```

[GDAL band numbers start at one. There is only one raster band in the DEM data we're using.]

Now that we have the raster band, we can use the `band.ReadRaster()` method to read the raw DEM data. This is what the `ReadRaster()` method looks like:

```
ReadRaster(x, y, width, height, dWidth, dHeight, pixelType)
```

Where:

- `x` is the number of pixels from the left side of the raster band to the left side of the portion of the band to read from
- `y` is the number of pixels from the top of the raster band to the top of the portion of the band to read from
- `width` is the number of pixels across to read
- `height` is the number of pixels down to read
- `dWidth` is the width of the resulting data
- `dHeight` is the height of the resulting data
- `pixelType` is a constant defining how many bytes of data there are for each pixel value, and how that data is to be interpreted

[Normally, you would set `dWidth` and `dHeight` to the same value as `width` and `height`; if you don't do this, the raster data will be scaled up or down when it is read.]

The `ReadRaster()` method returns a string containing the raster data as a raw sequence of bytes. You can then read the individual values from this string using the `struct` standard library module:

```
values = struct.unpack("<" + ("h" * width), data)
```

Putting all this together, we can use GDAL to open the raster datafile and read all the pixel values within the bounding box surrounding New Zealand:

```
import sys, struct
from osgeo import gdal
from osgeo import gdalconst

minLat  = -48
maxLat  = -33
minLong = 165
maxLong = 179

dataset = gdal.Open("l10g")
band = dataset.GetRasterBand(1)

t = dataset.GetGeoTransform()
success,tInverse = gdal.InvGeoTransform(t)
if not success:
    print "Failed!"
    sys.exit(1)

x1,y1 = gdal.ApplyGeoTransform(tInverse, minLong, minLat)
```

```
    x2,y2 = gdal.ApplyGeoTransform(tInverse, maxLong, maxLat)

    minX = int(min(x1, x2))
    maxX = int(max(x1, x2))
    minY = int(min(y1, y2))
    maxY = int(max(y1, y2))

    width = (maxX - minX) + 1
    fmt = "<" + ("h" * width)

    for y in range(minY, maxY+1):
        scanline = band.ReadRaster(minX, y,width, 1,
                                   width, 1,
                                   gdalconst.GDT_Int16)
        values = struct.unpack(fmt, scanline)

        for value in values:
        ...
```

Don't forget to add a directory path to the `gdal.Open()` statement if you placed the `l10g` file in a different directory.

Let's replace the `...` with some code that does something useful with the pixel values. We will calculate a histogram:

```
    histogram = {} # Maps height to # pixels with that height.

    ...

    for value in values:
        try:
            histogram[value] += 1
        except KeyError:
            histogram[value] = 1

    for height in sorted(histogram.keys()):
        print height,histogram[height]
```

If you run this, you will see a list of heights (in meters) and how many pixels there are at that height:

```
-500 2607581
1 6641
2 909
3 1628
...
3097 1
3119 2
3173 1
```

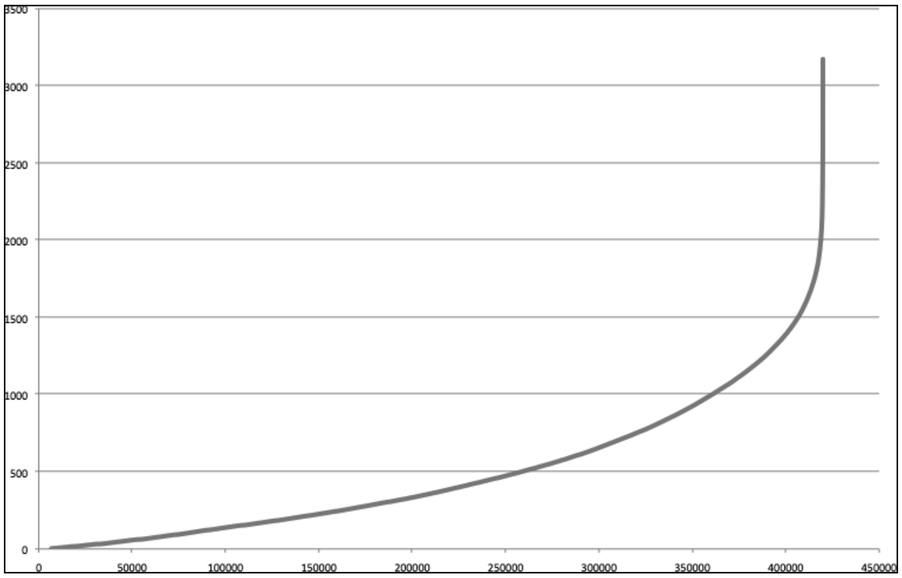
This reveals one final problem—there are a large number of pixels with a value of `-500`. What is going on here? Clearly `-500` is not a valid height value. The GLOBE documentation explains:

> *Every tile contains values of -500 for oceans, with no values between -500 and the minimum value for land noted here.*

So, all those points with a value of `-500` represents pixels over the ocean. Fortunately, it is easy to exclude these; every raster file includes the concept of a **no data value** that is used for pixels without valid data. GDAL includes the `GetNoDataValue()` method that allows us to exclude these pixels:

```
for value in values:
    if value != band.GetNoDataValue():
        try:
            histogram[value] += 1
        except KeyError:
            histogram[value] = 1
```

This finally gives us a histogram of the heights across New Zealand. You could create a graph using this data if you wished. For example, the following chart shows the total number of pixels at or below a given height:



Changing datums and projections

If you can remember from *Chapter 2*, a **datum** is a mathematical model of the Earth's shape, while a **projection** is a way of translating points on the Earth's surface into points on a two-dimensional map. There are a large number of available datums and projections—whenever you are working with geo-spatial data, you must know which datum and which projection (if any) your data uses. If you are combining data from multiple sources, you will often have to change your geo-spatial data from one datum to another, or from one projection to another.

Task: Change projections to combine Shapefiles using geographic and UTM coordinates

Here, we will work with two Shapefiles that have different projections. We haven't yet encountered any geo-spatial data that uses a projection—all the data we've seen so far uses geographic (unprojected) latitude and longitude values. So, let's start by downloading some geo-spatial data in UTM (Universal Transverse Mercator) projection.

The WebGIS website (`http://webgis.com`) provides Shapefiles describing land-use and land-cover, called LULC datafiles. For this example, we will download a Shapefile for southern Florida (Dade County, to be exact) which uses the Universal Transverse Mercator projection.

You can download this Shapefile from the following URL:

`http://webgis.com/MAPS/fl/lulcutm/miami.zip`

The uncompressed directory contains the Shapefile, called `miami.shp`, along with a `datum_reference.txt` file describing the Shapefile's coordinate system. This file tells us the following:

```
The LULC shape file was generated from the original USGS GIRAS LULC
file by Lakes Environmental Software.
Datum: NAD83
Projection: UTM
Zone: 17
Data collection date by U.S.G.S.: 1972
Reference: http://edcwww.cr.usgs.gov/products/landcover/lulc.html
```

So, this particular Shapefile uses UTM Zone 17 projection, and a datum of NAD83.

Let's take a second Shapefile, this time in geographic coordinates. We'll use the GSHHS shoreline database, which uses the WGS84 datum and geographic (latitude/longitude) coordinates.

> You don't need to download the GSHHS database for this example; while we will display a map overlaying the LULC data over the top of the GSHHS data, you only need the LULC Shapefile to complete this recipe. Drawing maps such as the one shown below will be covered in *Chapter 8*.

Combining these two Shapefiles as they are would be impossible—the LULC Shapefile has coordinates measured in UTM (that is, in meters from a given reference line), while the GSHHS Shapefile has coordinates in latitude and longitude values (in decimal degrees):

```
LULC:  x=485719.47, y=2783420.62
       x=485779.49,y=2783380.63
       x=486129.65, y=2783010.66
       ...

GSHHS: x=180.0000,y=68.9938
       x=180.0000,y=65.0338
       x=179.9984, y=65.0337
       ...
```

Before we can combine these two Shapefiles, we first have to convert them to use the same projection. We'll do this by converting the LULC Shapefile from UTM-17 to geographic projection. Doing this requires us to define a **coordinate transformation** and then apply that transformation to each of the features in the Shapefile.

Here is how you can define a coordinate transformation using OGR:

```
from osgeo import osr

srcProjection = osr.SpatialReference()
srcProjection.SetUTM(17)

dstProjection = osr.SpatialReference()
dstProjection.SetWellKnownGeogCS('WGS84') # Lat/long.

transform = osr.CoordinateTransformation(srcProjection,
                                         dstProjection)
```

Using this transformation, we can transform each of the features in the Shapefile from UTM projection back into geographic coordinates:

```
for i in range(layer.GetFeatureCount()):
    feature = layer.GetFeature(i)
    geometry = feature.GetGeometryRef()
    geometry.Transform(transform)
    ...
```

Putting all this together with the techniques we explored earlier for copying the features from one Shapefile to another, we end up with the following complete program:

```
# changeProjection.py

import os, os.path, shutil
from osgeo import ogr
from osgeo import osr
from osgeo import gdal

# Define the source and destination projections, and a
# transformation object to convert from one to the other.

srcProjection = osr.SpatialReference()
srcProjection.SetUTM(17)

dstProjection = osr.SpatialReference()
dstProjection.SetWellKnownGeogCS('WGS84') # Lat/long.

transform = osr.CoordinateTransformation(srcProjection,
                                         dstProjection)

# Open the source shapefile.

srcFile = ogr.Open("miami/miami.shp")
srcLayer = srcFile.GetLayer(0)

# Create the dest shapefile, and give it the new projection.

if os.path.exists("miami-reprojected"):
    shutil.rmtree("miami-reprojected")
os.mkdir("miami-reprojected")

driver = ogr.GetDriverByName("ESRI Shapefile")
dstPath = os.path.join("miami-reprojected", "miami.shp")
dstFile = driver.CreateDataSource(dstPath)
dstLayer = dstFile.CreateLayer("layer", dstProjection)

# Reproject each feature in turn.

for i in range(srcLayer.GetFeatureCount()):
    feature = srcLayer.GetFeature(i)
    geometry = feature.GetGeometryRef()
```

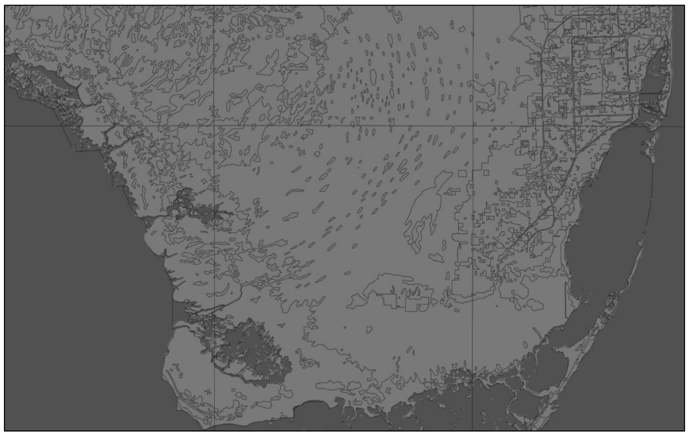
```
        newGeometry = geometry.Clone()
        newGeometry.Transform(transform)

        feature = ogr.Feature(dstLayer.GetLayerDefn())
        feature.SetGeometry(newGeometry)
        dstLayer.CreateFeature(feature)
        feature.Destroy()

    # All done.

    srcFile.Destroy()
    dstFile.Destroy()
```

Note that this example doesn't copy field values into the new Shapefile; if your Shapefile has metadata, you will want to copy the fields across as you create each new feature. Also, the above code assumes that the `miami.shp` Shapefile has been placed into a `miami` sub-directory; you'll need to change the `ogr.Open()` statement to use the appropriate path name if you've stored this Shapefile in a different place.

After running this program over the `miami.shp` Shapefile, the coordinates for all the features in the Shapefile will have been converted from UTM-17 into geographic coordinates:

```
Before reprojection:  x=485719.47, y=2783420.62
                      x=485779.49, y=2783380.63
                      x=486129.65, y=2783010.66
                      ...

  After reprojection: x=-81.1417, y=25.1668
                      x=-81.1411, y=25.1664
                      x=-81.1376, y=25.1631
                      ...
```

To see that this worked, let's draw a map showing the reprojected LULC data on top of the GSHHS shoreline database:



Both Shapefiles now use geographic coordinates, and as you can see the coastlines match exactly.

[If you have been watching closely, you may have noticed that the LULC data is using the NAD83 datum, while the GSHHS data and our reprojected version of the LULC data both use the WGS84 datum. We can do this without error because the two datums are identical for points within North America.]

Task: Change datums to allow older and newer TIGER data to be combined

For this example, we will need to obtain some geo-spatial data that uses the NAD27 datum. This datum dates back to 1927, and was commonly used for North American geo-spatial analysis up until the 1980s when it was replaced by NAD83.

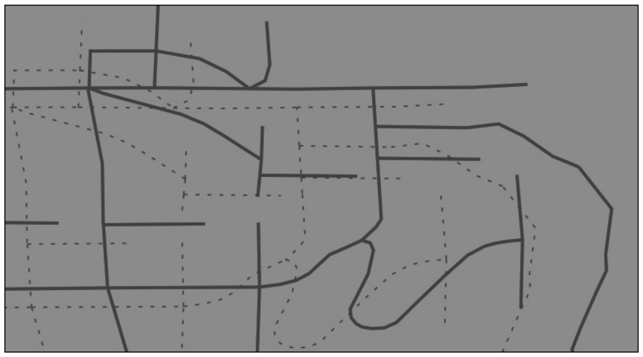
ESRI makes available a set of TIGER/Line files from the 2000 US census, converted into Shapefile format. These files can be downloaded from:

`http://esri.com/data/download/census2000-tigerline/index.html`

For the 2000 census data, the TIGER/Line files were all in NAD83 with the exception of Alaska, which used the older NAD27 datum. So, we can use this site to download a Shapefile containing features in NAD27. Go to the above site, click on the **Preview and Download** hyperlink, and then choose `Alaska` from the drop-down menu. Select the `Line Features - Roads` layer, then click on the **Submit Selection** button.

This data is divided up into individual counties. Click on the checkbox beside **Anchorage**, then click on the **Proceed to Download** button to download the Shapefile containing road details in Anchorage. The resulting Shapefile will be named `tgr02020lkA.shp`, and will be in a directory called `lkA02020`.

As described on the website, this data uses the NAD27 datum. If we were to assume this Shapefile used the WSG83 datum, all the features would be in the wrong place:



The heavy lines indicate where the features would appear if they were plotted using the incorrect WGS84 datum, while the thin dashed lines show where the features should really appear.

To make the features appear in the correct place, and to be able to combine these features with other features that use the WGS84 datum, we need to convert the Shapefile to use WGS84. Changing a Shapefile from one datum to another requires the same basic process we used earlier to change a Shapefile from one projection to another: first, you choose the source and destination datums, and define a coordinate transformation to convert from one to the other:

```
srcDatum = osr.SpatialReference()
srcDatum.SetWellKnownGeogCS('NAD27')

dstDatum = osr.SpatialReference()
dstDatum.SetWellKnownGeogCS('WGS84')

transform = osr.CoordinateTransformation(srcDatum, dstDatum)
```

You then process each feature in the Shapefile, transforming the feature's geometry using the coordinate transformation:

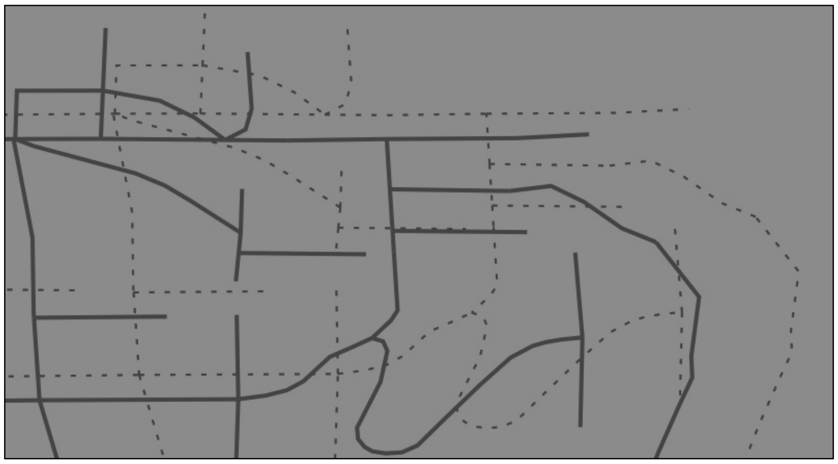
```
for i in range(srcLayer.GetFeatureCount()):
    feature = srcLayer.GetFeature(i)
    geometry = feature.GetGeometryRef()
    geometry.Transform(transform)
    ...
```

Here is the complete Python program to convert the `lkA02020` Shapefile from the NAD27 datum to WGS84:

```
# changeDatum.py

import os, os.path, shutil
from osgeo import ogr
from osgeo import osr
from osgeo import gdal

# Define the source and destination datums, and a
# transformation object to convert from one to the other.

srcDatum = osr.SpatialReference()
srcDatum.SetWellKnownGeogCS('NAD27')

dstDatum = osr.SpatialReference()
dstDatum.SetWellKnownGeogCS('WGS84')

transform = osr.CoordinateTransformation(srcDatum, dstDatum)

# Open the source shapefile.

srcFile = ogr.Open("lkA02020/tgr02020lkA.shp")
srcLayer = srcFile.GetLayer(0)

# Create the dest shapefile, and give it the new projection.

if os.path.exists("lkA-reprojected"):
    shutil.rmtree("lkA-reprojected")
os.mkdir("lkA-reprojected")

driver = ogr.GetDriverByName("ESRI Shapefile")
dstPath = os.path.join("lkA-reprojected", "lkA02020.shp")
dstFile = driver.CreateDataSource(dstPath)
dstLayer = dstFile.CreateLayer("layer", dstDatum)

# Reproject each feature in turn.

for i in range(srcLayer.GetFeatureCount()):
    feature = srcLayer.GetFeature(i)
    geometry = feature.GetGeometryRef()

    newGeometry = geometry.Clone()
    newGeometry.Transform(transform)

    feature = ogr.Feature(dstLayer.GetLayerDefn())
    feature.SetGeometry(newGeometry)
    dstLayer.CreateFeature(feature)
    feature.Destroy()

# All done.

srcFile.Destroy()
dstFile.Destroy()
```

The above code assumes that the `lkA02020` folder is in the same directory as the Python script itself. If you've placed this folder somewhere else, you'll need to change the `ogr.Open()` statement to use the appropriate directory path.

If we now plot the reprojected features using the WGS84 datum, the features will appear in the correct place:



The thin dashed lines indicate where the original projection would have placed the features, while the heavy lines show the correct positions using the reprojected data.

Representing and storing geo-spatial data

While geo-spatial data is often supplied in the form of vector-format files such as Shapefiles, there are situations where Shapefiles are unsuitable or inefficient. One such situation is where you need to take geo-spatial data from one library and use it in a different library. For example, imagine that you have read a set of geometries out of a Shapefile and want to store them in a database, or work with them using the Shapely library. Because the different Python libraries all use their own private classes to represent geo-spatial data, you can't just take an OGR `Geometry` object and pass it to Shapely, or use a GDAL `SpatialReference` object to define the datum and projection to use for data stored in a database.

In these situations, you need to have an independent format for representing and storing geo-spatial data that isn't limited to just one particular Python library. This format, the *lingua franca* for vector-format geo-spatial data, is called **Well-Known Text** or **WKT**.

WKT is a compact text-based description of a geo-spatial object such as a point, a line, or a polygon. For example, here is a geometry defining the boundary of the Vatican City in the World Borders Dataset, converted into a WKT string:

```
POLYGON ((12.445090330888604 41.90311752178485,
12.451653339580503 41.907989033391232,
12.456660170953796 41.901426024699163,
12.445090330888604 41.90311752178485))
```

As you can see, the WKT string contains a straightforward text description of a geometry—in this case, a polygon consisting of four x,y coordinates. Obviously, WKT text strings can be far more complex than this, containing many thousands of points and storing multipolygons and collections of different geometries. No matter how complex the geometry is, it can still be represented as a simple text string.

There is an equivalent binary format called **Well-Known Binary** (**WKB**) that stores the same information as binary data. WKB is often used to store geo-spatial data into a database.

WKT strings can also be used to represent a **spatial reference** encompassing a projection, a datum, and/or a coordinate system. For example, here is an `osgeo.osr.SpatialReference` object representing a geographic coordinate system using the WGS84 datum, converted into a WKT string:

```
GEOGCS["WGS 84",DATUM["WGS_1984",SPHEROID["WGS
84",6378137,298.257223563,AUTHORITY["EPSG","7030"]],TOWGS84[0,0,0,0,0,0,
0],AUTHORITY["EPSG","6326"]],PRIMEM["Greenwich",0,AUTHORITY["EPSG","8901
"]],UNIT["degree",0.0174532925199433,AUTHORITY["EPSG","9108"]],AUTHORITY
["EPSG","4326"]]
```

As with geometry representations, spatial references in WKT format can be used to pass a spatial reference from one Python library to another.

Task: Calculate the border between Thailand and Myanmar

In this recipe, we will make use of the World Borders Dataset to obtain polygons defining the borders of Thailand and Myanmar. We will then transfer these polygons into Shapely, and use Shapely's capabilities to calculate the common border between these two countries.

If you haven't already done so, download the World Borders Dataset from the Thematic Mapping website:

`http://thematicmapping.org/downloads/world_borders.php`

The World Borders Dataset conveniently includes ISO 3166 two-character country codes for each feature, so we can identify the features corresponding to Thailand and Myanmar as we read through the Shapefile:

```
import osgeo.ogr

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

for i in range(layer.GetFeatureCount()):
    feature = layer.GetFeature(i)
    if feature.GetField("ISO2") == "TH":
        ...
    elif feature.GetField("ISO2") == "MM":
        ...
```

This code assumes that you have placed the `TM_WORLD_BORDERS-0.3.shp` Shapefile in the same directory as the Python script. If you've placed it into a different directory, you'll need to adjust the `osgeo.ogr.Open()` statement to match.

Once we have identified the features we want, it is easy to extract the features' geometries as WKT strings:

```
geometry = feature.GetGeometryRef()
wkt = geometry.ExportToWkt()
```

We can then convert these to Shapely geometry objects using the `shapely.wkt` module:

```
import shapely.wkt
...
border = shapely.wkt.loads(wkt)
```

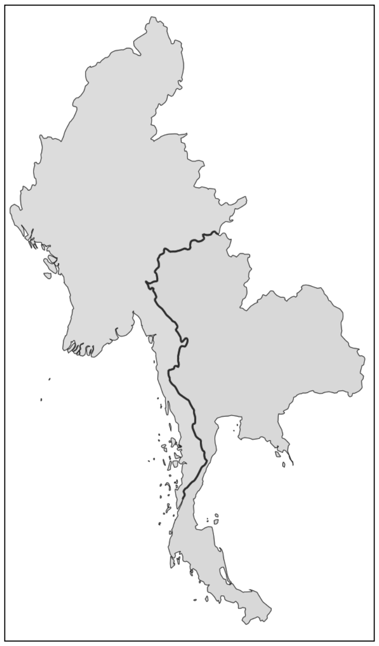
Now that we have the objects in Shapely, we can use Shapely's computational geometry capabilities to calculate the common border between these two countries:

```
commonBorder = thailandBorder.intersection(myanmarBorder)
```

The result will be a LineString (or a MultiLineString if the border is broken up into more than one part). If we wanted to, we could then convert this Shapely object back into an OGR geometry, and save it into a Shapefile again:

```
wkt = shapely.wkt.dumps(commonBorder)

feature = osgeo.ogr.Feature(dstLayer.GetLayerDefn())
feature.SetGeometry(osgeo.ogr.CreateGeometryFromWkt(wkt))
dstLayer.CreateFeature(feature)
feature.Destroy()
```

With the common border saved into a Shapefile, we can display the results as a map:



The contents of the `common-border/border.shp` Shapefile is represented by the heavy line along the countries' common borders.

Here is the entire program used to calculate this common border:

```
# calcCommonBorders.py

import os,os.path,shutil

import osgeo.ogr
import shapely.wkt

# Load the thai and myanmar polygons from the world borders
# dataset.

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

thailand = None
myanmar = None

for i in range(layer.GetFeatureCount()):
    feature = layer.GetFeature(i)
    if feature.GetField("ISO2") == "TH":
        geometry = feature.GetGeometryRef()
        thailand = shapely.wkt.loads(geometry.ExportToWkt())
        elif feature.GetField("ISO2") == "MM":
            geometry = feature.GetGeometryRef()
            myanmar = shapely.wkt.loads(geometry.ExportToWkt())

    # Calculate the common border.

    commonBorder = thailand.intersection(myanmar)

    # Save the common border into a new shapefile.

    if os.path.exists("common-border"):
        shutil.rmtree("common-border")
    os.mkdir("common-border")

    spatialReference = osgeo.osr.SpatialReference()
    spatialReference.SetWellKnownGeogCS('WGS84')

    driver = osgeo.ogr.GetDriverByName("ESRI Shapefile")
    dstPath = os.path.join("common-border", "border.shp")
    dstFile = driver.CreateDataSource(dstPath)
    dstLayer = dstFile.CreateLayer("layer", spatialReference)

    wkt = shapely.wkt.dumps(commonBorder)

    feature = osgeo.ogr.Feature(dstLayer.GetLayerDefn())
    feature.SetGeometry(osgeo.ogr.CreateGeometryFromWkt(wkt))
    dstLayer.CreateFeature(feature)
    feature.Destroy()

    dstFile.Destroy()
```

If you've placed your `TM_WORLD_BORDERS-0.3.shp` Shapefile into a different directory, change the `osgeo.ogr.Open()` statement to include a suitable directory path.

We will use this Shapefile later in this chapter to calculate the length of the Thai-Myanmar border, so make sure you generate and keep a copy of the `common-borders/border.shp` Shapefile.

Task: Save geometries into a text file

WKT is not only useful for transferring geometries from one Python library to another. It can also be a useful way of *storing* geo-spatial data without having to deal with the complexity and constraints imposed by using Shapefiles.

In this example, we will read a set of polygons from the World Borders Dataset, convert them to WKT format, and save them as text files:

```
# saveAsText.py

import os,os.path,shutil

import osgeo.ogr

if os.path.exists("country-wkt-files"):
    shutil.rmtree("country-wkt-files")
os.mkdir("country-wkt-files")

shapefile = osgeo.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()

    f = file(os.path.join("country-wkt-files",
                          name + ".txt"), "w")
    f.write(geometry.ExportToWkt())
    f.close()
```

As usual, you'll need to change the `osgeo.ogr.Open()` statement to include a directory path if you've stored the Shapefile in a different directory.

You might be wondering why you want to do this, rather than creating a Shapefile to store your geo-spatial data. Well, Shapefiles are limited in that all the features in a single Shapefile must have the same geometry type. Also, the complexity of setting up metadata and saving geometries can be overkill for some applications. Sometimes, dealing with plain text is just easier.

Working with Shapely geometries

Shapely is a very capable library for performing various calculations on geo-spatial data. Let's put it through its paces with a complex, real-world problem.

Task: Identify parks in or near urban areas

The U.S. Census Bureau makes available a Shapefile containing something called **Core Based Statistical Areas** (CBSAs), which are polygons defining urban areas with a population of 10,000 or more. At the same time, the GNIS website provides lists of placenames and other details. Using these two datasources, we will identify any parks within or close to an urban area.

Because of the volume of data we are potentially dealing with, we will limit our search to California. Feel free to download the larger data sets if you want, though you will have to optimize the code or your program will take a *very* long time to check all the CBSA polygon/placename combinations.

1. Let's start by downloading the necessary data. Go to the TIGER website at `http://census.gov/geo/www/tiger`
2. Click on the **2009 TIGER/Line Shapefiles Main Page** link, then follow the **Download the 2009 TIGER/Line Shapefiles now** link.
3. Choose **California** from the pop-up menu on the right, and click on **Submit**. A list of the California Shapefiles will be displayed; the Shapefile you want is labelled **Metropolitan/Micropolitan Statistical Area**. Click on this link, and you will download a file named `tl_2009_06_cbsa.zip`. Once the file has downloaded, uncompress it and place the resulting Shapefile into a convenient location so that you can work with it.
4. You now need to download the GNIS placename data for California. Go to the GNIS website:

   `http://geonames.usgs.gov/domestic`
5. Click on the **Download Domestic Names** hyperlink, and then choose **California** from the pop-up menu. You will be prompted to save the `CA_Features_XXX.zip` file. Do so, then decompress it and place the resulting `CA_Features_XXX.txt` file into a convenient place.

The `XXX` in the above file name is a date stamp, and will vary depending on when you download the data. Just remember the name of the file as you'll need to refer to it in your source code.

6. We're now ready to write the code. Let's start by reading through the CBSA urban area Shapefile and extracting the polygons that define the boundary of each urban area:

```
shapefile = osgeo.ogr.Open("tl_2009_06_cbsa.shp")
layer = shapefile.GetLayer(0)
```

```
for i in range(layer.GetFeatureCount()):
    feature = layer.GetFeature(i)
    geometry = feature.GetGeometryRef()
    ...
```

[Make sure you add directory paths to your `osgeo.ogr.Open()` statement (and to the `file()` statement below) to match where you've placed these files.]

7. Using what we learned in the previous section, we can convert this geometry into a Shapely object so that we can work with it:

   ```
   wkt = geometry.ExportToWkt()
   shape = shapely.wkt.loads(wkt)
   ```

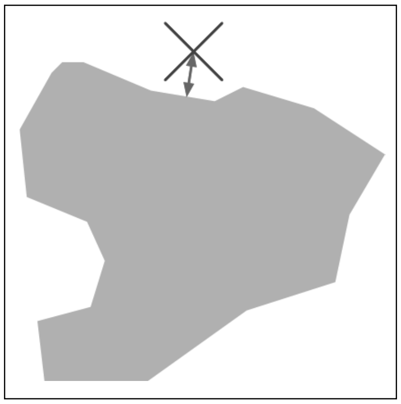
8. Next, we need to scan through the `CA_Features_XXX.txt` file to identify the features marked as a park. For each of these features, we want to extract the name of the feature and its associated latitude and longitude. Here's how we might do this:

   ```
   f = file("CA_Features_XXX.txt", "r")
   for line in f.readlines():
       chunks = line.rstrip().split("|")
       if chunks[2] == "Park":
           name = chunks[1]
           latitude = float(chunks[9])
           longitude = float(chunks[10])
           ...
   ```

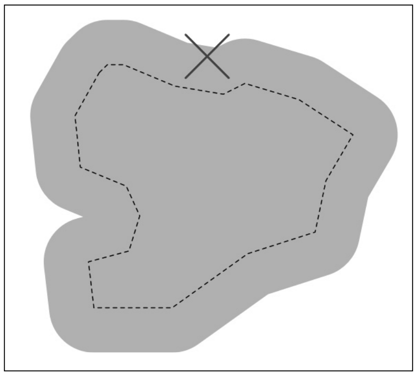
[Remember that the GNIS placename database is a *pipe-delimited* text file. That's why we have to split the line up using `line.rstrip().split("|")`.]

9. Now comes the fun part—we need to figure out which parks are within or close to each urban area. There are two ways we could do this, either of which will work:

    ◦ We could use the `shape.distance()` method to calculate the distance between the shape and a `Point` object representing the park's location:



    ◦ We could *dilate* the polygon using the `shape.buffer()` method, and then see if the resulting polygon contained the desired point:



The second option is faster when dealing with a large number of points as we can pre-calculate the dilated polygons and then use them to compare against each point in turn. Let's take this option:

```
# findNearbyParks.py

import osgeo.ogr
import shapely.geometry
import shapely.wkt

MAX_DISTANCE = 0.1 # Angular distance; approx 10 km.

print "Loading urban areas..."

urbanAreas = {} # Maps area name to Shapely polygon.

shapefile = osgeo.ogr.Open("tl_2009_06_cbsa.shp")
layer = shapefile.GetLayer(0)

for i in range(layer.GetFeatureCount()):
    feature = layer.GetFeature(i)
    name = feature.GetField("NAME")
    geometry = feature.GetGeometryRef()
    shape = shapely.wkt.loads(geometry.ExportToWkt())
    dilatedShape = shape.buffer(MAX_DISTANCE)
    urbanAreas[name] = dilatedShape

print "Checking parks..."

f = file("CA_Features_XXX.txt", "r")
for line in f.readlines():
    chunks = line.rstrip().split("|")
    if chunks[2] == "Park":
        parkName = chunks[1]
        latitude = float(chunks[9])
        longitude = float(chunks[10])

        pt = shapely.geometry.Point(longitude, latitude)

        for urbanName,urbanArea in urbanAreas.items():
            if urbanArea.contains(pt):
                print parkName + " is in or near " + urbanName
f.close()
```

Don't forget to change the name of the `CA_Features_XXX.txt` file to match the actual name of the file you downloaded. You may also need to change the path names to the `tl_2009_06_CBSA.shp` file and the `CA_Features` file if you placed them in a different directory.

If you run this program, you will get a master list of all the parks that are in or close to an urban area:

```
% python findNearbyParks.py
Loading urban areas...
Checking parks...
```

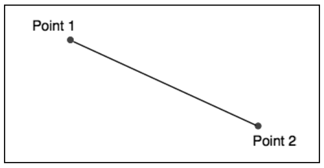
```
Imperial National Wildlife Refuge is in or near El Centro, CA
TwinLakesStateBeach is in or near Santa Cruz-Watsonville, CA
AdmiralWilliamStandleyState Recreation Area is in or near Ukiah, CA
Agate Beach County Park is in or near San Francisco-Oakland-Fremont, CA
...
```

Note that our program uses **angular distances** to decide if a park is in or near a given urban area. We looked at angular distances in *Chapter 2*. An angular distance is the angle (in decimal degrees) between two rays going out from the center of the Earth to the Earth's surface. Because a degree of angular measurement (at least for the latitudes we are dealing with here) roughly equals 100 km on the Earth's surface, an angular measurement of 0.1 roughly equals a real distance of 10 km.

Using angular measurements makes the distance calculation easy and quick to calculate, though it doesn't give an exact distance on the Earth's surface. If your application requires exact distances, you could start by using an angular distance to filter out the features obviously too far away, and then obtain an exact result for the remaining features by calculating the point on the polygon's boundary that is closest to the desired point, and then calculating the linear distance between the two points. You would then discard the points that exceed your desired exact linear distance. Implementing this would be an interesting challenge, though not one we will examine in this book.

Converting and standardizing units of geometry and distance

Imagine that you have two points on the Earth's surface with a straight line drawn between them:



Each point can be described as a coordinate using some arbitrary coordinate system (for example, using latitude and longitude values), while the length of the straight line could be described as the distance between the two points.

Given any two coordinates, it is possible to calculate the distance between them. Conversely, you can start with one coordinate, a desired distance and a direction, and then calculate the coordinates for the other point.

Of course, because the Earth's surface is not flat, we aren't really dealing with straight lines at all. Rather, we are calculating geodetic or **Great Circle** distances across the surface of the Earth.

The `pyproj` Python library allows you to perform these types of calculations for any given datum. You can also use `pyproj` to convert from projected coordinates back to geographic coordinates, and *vice versa*, allowing you to perform these sorts of calculations for any desired datum, coordinate system, and projection.

Ultimately, a geometry such as a line or a polygon consists of nothing more than a list of connected points. This means that, using the above process, you can calculate the geodetic distance between each of the points in any polygon and total the results to get the actual length for any geometry. Let's use this knowledge to solve a real-world problem.

Task: Calculate the length of the Thai-Myanmar border

To solve this problem, we will make use of the `common-borders/border.shp` Shapefile we created earlier. This Shapefile contains a single feature, which is a LineString defining the border between the two countries. Let's start by taking a look at the individual line segments that make up this feature's geometry:

```
import os.path
import osgeo.ogr

def getLineSegmentsFromGeometry(geometry):
    segments = []
    if geometry.GetPointCount() > 0:
        segment = []
        for i in range(geometry.GetPointCount()):
            segment.append(geometry.GetPoint_2D(i))
        segments.append(segment)
    for i in range(geometry.GetGeometryCount()):
        subGeometry = geometry.GetGeometryRef(i)
        segments.extend(
getLineSegmentsFromGeometry(subGeometry))
    return segments

filename = os.path.join("common-border", "border.shp")
shapefile = osgeo.ogr.Open(filename)
layer = shapefile.GetLayer(0)
feature = layer.GetFeature(0)
geometry = feature.GetGeometryRef()

segments = getLineSegmentsFromGeometry(geometry)

print segments
```

[Don't forget to change the `os.path.join()` statement to match the location of your `border.shp` Shapefile.]

Note that we use a recursive function, `getLineSegmentsFromGeometry()`, to pull the individual coordinates for each line segment out of the geometry. Because geometries are recursive data structures, we have to pull out the individual line segments before we can work with them.

Running this program produces a long list of points that make up the various line segments defining the border between these two countries:

```
% python calcBorderLength.py
[[(100.08132200000006, 20.348840999999936),
(100.08943199999999, 20.347217999999941)],
[(100.08943199999999, 20.347217999999941),
(100.0913700000001, 20.348606000000075)], ...]
```

Each line segment consists of a list of points—in this case, you'll notice that each segment has only two points—and if you look closely you will notice that each segment starts at the same point as the previous segment ended. There are a total of 459 segments defining the border between Thailand and Myanmar—that is, 459 point pairs that we can calculate the geodetic distance for.

[A geodetic distance is a distance measured on the surface of the Earth.]

Let's see how we can use `pyproj` to calculate the geodetic distance between any two points. We first create a `Geod` instance:

```
    geod = pyproj.Geod(ellps='WGS84')
```

`Geod` is the `pyproj` class that performs geodetic calculations. Note that we have to provide it with details of the datum used to describe the shape of the Earth. Once our `Geod` instance has been set up, we can calculate the geodetic distance between any two points by calling `geod.inv()`, the *inverse geodetic transformation* method:

```
    angle1,angle2,distance = geod.inv(long1, lat1, long2, lat2)
```

`angle1` will be the angle from the first point to the second, measured in decimal degrees; `angle2` will be the angle from the second point back to the first (again in degrees); and `distance` will be the Great Circle distance between the two points, in meters.

Using this, we can iterate over the line segments, calculate the distance from one point to another, and total up all the distances to obtain the total length of the border:

```
geod = pyproj.Geod(ellps='WGS84')

totLength = 0.0
for segment in segments:
    for i in range(len(segment)-1):
        pt1 = segment[i]
        pt2 = segment[i+1]

        long1,lat1 = pt1
        long2,lat2 = pt2

        angle1,angle2,distance = geod.inv(long1, lat1,
        long2, lat2)
        totLength += distance
```

Upon completion, `totLength` will be the total length of the border, in meters.

Putting all this together, we end up with a complete Python program to read the `border.shp` Shapefile, and calculate and then display the total length of the common border:

```
# calcBorderLength.py

import os.path
import osgeo.ogr
import pyproj

def getLineSegmentsFromGeometry(geometry):
    segments = []
    if geometry.GetPointCount() > 0:
        segment = []
        for i in range(geometry.GetPointCount()):
            segment.append(geometry.GetPoint_2D(i))
        segments.append(segment)
    for i in range(geometry.GetGeometryCount()):
        subGeometry = geometry.GetGeometryRef(i)
        segments.extend(
            getLineSegmentsFromGeometry(subGeometry))
    return segments

filename = os.path.join("common-border", "border.shp")
shapefile = osgeo.ogr.Open(filename)
layer = shapefile.GetLayer(0)
feature = layer.GetFeature(0)
geometry = feature.GetGeometryRef()
segments = getLineSegmentsFromGeometry(geometry)
```

```
    geod = pyproj.Geod(ellps='WGS84')

    totLength = 0.0
    for segment in segments:
        for i in range(len(segment)-1):
            pt1 = segment[i]
            pt2 = segment[i+1]

            long1,lat1 = pt1
            long2,lat2 = pt2

            angle1,angle2,distance = geod.inv(long1, lat1,
                                              long2, lat2)
            totLength += distance

    print "Total border length = %0.2f km" % (totLength/1000)
```

Running this tells us the total calculated length of the Thai-Myanmar border:

```
% python calcBorderLength.py
Total border length = 1730.55 km
```

In this program, we have assumed that the Shapefile is in geographic coordinates using the WGS84 ellipsoid, and only contains a single feature. Let's extend our program to deal with any supplied projection and datum, and at the same time process *all* the features in the Shapefile rather than just the first. This will make our program more flexible, and allow it to work with any arbitrary Shapefile rather than just the common-border Shapefile we created earlier.

Let's deal with the projection and datum first. We could change the projection and datum for our Shapefile before we process it, just as we did with the `LULC` and `lkA02020` Shapefiles earlier in this chapter. That would work, but it would require us to create a temporary Shapefile just to calculate the length, which isn't very efficient. Instead, let's make use of `pyproj` directly to reproject the Shapefile's contents back into geographic coordinates if necessary. We can do this by querying the Shapefile's spatial reference:

```
    shapefile = ogr.Open(filename)
    layer = shapefile.GetLayer(0)
    spatialRef = layer.GetSpatialRef()
    if spatialRef == None:
        print "Shapefile has no spatial reference, using WGS84."
        spatialRef = osr.SpatialReference()
        spatialRef.SetWellKnownGeogCS('WGS84')
```

Once we have the spatial reference, we can see if the spatial reference is projected, and if so use `pyproj` to turn the projected coordinates back into lat/long values again, like this:

```
if spatialRef.IsProjected():
    # Convert projected coordinates back to lat/long values.
srcProj = pyproj.Proj(spatialRef.ExportToProj4())
dstProj = pyproj.Proj(proj='longlat', ellps='WGS84',
 datum='WGS84')
...
long,lat = pyproj.transform(srcProj, dstProj, x, y)
```

Using this, we can rewrite our program to accept data using any projection and datum. At the same time, we'll change it to calculate the overall length of every feature in the file, rather than just the first, and also to accept the name of the Shapefile from the command line. Finally, we'll add some error-checking. Let's call the results `calcFeatureLengths.py`.

We'll start by copying the `getLineSegmentsFromGeometry()` function we used earlier:

```
import sys
from osgeo import ogr, osr
import pyproj

def getLineSegmentsFromGeometry(geometry):
    segments = []
    if geometry.GetPointCount() > 0:
        segment = []
        for i in range(geometry.GetPointCount()):
            segment.append(geometry.GetPoint_2D(i))
        segments.append(segment)
    for i in range(geometry.GetGeometryCount()):
        subGeometry = geometry.GetGeometryRef(i)
        segments.extend(
            getLineSegmentsFromGeometry(subGeometry))
    return segments
```

Next, we'll get the name of the Shapefile to open from the command line:

```
if len(sys.argv) != 2:
    print "Usage: calcFeatureLengths.py <shapefile>"
    sys.exit(1)

filename = sys.argv[1]
```

We'll then open the Shapefile and obtain its spatial reference, using the code we wrote earlier:

```
shapefile = ogr.Open(filename)
layer = shapefile.GetLayer(0)
spatialRef = layer.GetSpatialRef()
if spatialRef == None:
        print "Shapefile lacks a spatial reference, using WGS84."
        spatialRef = osr.SpatialReference()
        spatialRef.SetWellKnownGeogCS('WGS84')
```

We'll then get the source and destination projections, again using the code we wrote earlier. Note that we only need to do this if we're using projected coordinates:

```
    if spatialRef.IsProjected():
        srcProj = pyproj.Proj(spatialRef.ExportToProj4())
        dstProj = pyproj.Proj(proj='longlat', ellps='WGS84',
                              datum='WGS84')
```

We are now ready to start processing the Shapefile's features:

```
    for i in range(layer.GetFeatureCount()):
        feature = layer.GetFeature(i)
```

Now that we have the feature, we can borrow the code we used earlier to calculate the total length of that feature's line segments:

```
        geometry = feature.GetGeometryRef()
        segments = getLineSegmentsFromGeometry(geometry)

        geod = pyproj.Geod(ellps='WGS84')

        totLength = 0.0
        for segment in segments:
            for j in range(len(segment)-1):
                pt1 = segment[j]
                pt2 = segment[j+1]

                long1,lat1 = pt1
                long2,lat2 = pt2
```

The only difference is that we need to transform the coordinates back to WGS84 if we are using a projected coordinate system:

```
                if spatialRef.IsProjected():
                    long1,lat1 = pyproj.transform(srcProj,
                                                  dstProj,
                                                  long1, lat1)
                    long2,lat2 = pyproj.transform(srcProj,
                                                  dstProj,
                                                  long2, lat2)
```

We can then use `pyproj` to calculate the distance between the two points, as we did in our earlier example. This time, though, we'll wrap it in a `try...except` statement so that any failure to calculate the distance won't crash the program:

```
                try:
                    angle1,angle2,distance = geod.inv(long1, lat1,
                                                  long2, lat2)
            except ValueError:
                print "Unable to calculate distance from " \
                    + "%0.4f,%0.4f to %0.4f,%0.4f" \
                    % (long1, lat1, long2, lat2)
                distance = 0.0
            totLength += distance
```

[The `geod.inv()` call can raise a `ValueError` if the two coordinates are in a place where an angle can't be calculated—for example, if the two points are at the poles.]

And finally, we can print out the feature's total length, in kilometers:

```
        print "Total length of feature %d is %0.2f km" \
            % (i, totLength/1000)
```

This program can be run over any Shapefile. For example, you could use it to calculate the border length for every country in the world by running it over the World Borders Dataset:

```
% python calcFeatureLengths.py TM_WORLD_BORDERS-0.3.shp
Total length of feature 0 is 127.28 km
Total length of feature 1 is 7264.69 km
Total length of feature 2 is 2514.76 km
Total length of feature 3 is 968.86 km
Total length of feature 4 is 1158.92 km
Total length of feature 5 is 6549.53 km
Total length of feature 6 is 119.27 km
...
```

This program is an example of converting geometry coordinates into distances. Let's take a look at the inverse calculation: using distances to calculate new geometry coordinates.

Task: Find a point 132.7 kilometers west of Soshone, California

Using the `CA_Features_XXX.txt` file we downloaded earlier, it is possible to find the latitude and longitude of Shoshone, a small town in California east of Las Vegas:

```
f = file("CA_Features_20100607.txt", "r")
for line in f.readlines():
    chunks = line.rstrip().split("|")
    if chunks[1] == "Shoshone" and \
       chunks[2] == "Populated Place":
        latitude = float(chunks[9])
        longitude = float(chunks[10])
        ...
```

Given this coordinate, we can use `pyproj` to calculate the coordinate of a point a given distance away, at a given angle:

```
geod = pyproj.Geod(ellps="WGS84")
newLong,newLat,invAngle = geod.fwd(latitude, longitude,
                                   angle, distance)
```

For this task, we are given the desired distance and we know that the angle we want is "due west". `pyproj` uses azimuth angles, which are measured clockwise from North. Thus, due west would correspond to an angle of 270 degrees.

Putting all this together, we can calculate the coordinates of the desired point:

```
# findShoshone.py

import pyproj

distance = 132.7 * 1000
angle    = 270.0

f = file("CA_Features_20100607.txt", "r")
for line in f.readlines():
    chunks = line.rstrip().split("|")
    if chunks[1] == "Shoshone" and \
       chunks[2] == "Populated Place":
        latitude = float(chunks[9])
        longitude = float(chunks[10])

        geod = pyproj.Geod(ellps='WGS84')
        newLong,newLat,invAngle = geod.fwd(longitude,
                                           latitude,
                                           angle, distance)

        print "Shoshone is at %0.4f,%0.4f" % (latitude,
                                              longitude)
        print "The point %0.2f km west of Shoshone " \
            % (distance/1000.0) \
            + "is at %0.4f, %0.4f" % (newLat, newLong)

f.close()
```

Running this program gives us the answer we want:

```
% python findShoshone.py
Shoshone is at 35.9730,-116.2711
The point 132.70 km west of Shoshone is at 35.9640,
-117.7423
```

Exercises

If you are interested in exploring the techniques used in this chapter further, you might like to challenge yourself with the following tasks:

- *Change the "Calculate Bounding Box" calculation to exclude outlying islands.*

> **Hint**
> You can split each country's MultiPolygon into individual Polygon objects, and then check the area of each polygon to exclude those that are smaller than a given total value.

- *Use the World Borders Dataset to create a new Shapefile, where each country is represented by a single "Point" geometry containing the geographical middle of each country.*

> **Hint**
> You can start with the country bounding boxes we calculated earlier, and then simply calculate the midpoint using:
> ```
> midLat = (minLat + maxLat) / 2
> midLong = (minLong + maxLong) / 2
> ```
> This won't be exact, but it gives a reasonable mid-point value for you to use.

- *Extend the histogram example given above to only include height values that fall inside a selected country's outline.*

> **Hint**
> Implementing this in an efficient way can be difficult. A good approach would be to identify the bounding box for each of the polygons that make up the country's outline, and then iterate over the DEM coordinates within that bounding box. You could then check to see if a given coordinate is actually inside the country's outline using `polygon.contains(point)`, and only add the height to the histogram if the point is indeed within the country's outline.

- *Optimize the "identify nearby parks" example given earlier so that it can work quickly with larger data sets.*

Hint

One possibility might be to calculate the rectangular bounding box around each park, and then expand that bounding box north, south, east, and west by the desired angular distance. You could then quickly exclude all the points that aren't in that bounding box before making the time-consuming call to `polygon.contains(point)`.

- *Calculate the total length of the coastline of the United Kingdom.*

Hint

Remember that a country outline is a MultiPolygon, where each Polygon in the MultiPolygon represents a single island. You will need to extract the exterior ring from each of these individual island polygons, and calculate the total length of the line segments within that exterior ring. You can then total the length of each individual island to get the length of the entire country's coastline.

- *Design your own reusable library of geo-spatial functions that build on OGR, GDAL, Shapely, and pyproj to perform common operations such as those discussed in this chapter.*

Hint

Writing your own reusable library modules is a common programming tactic. Think about the various tasks we have solved in this chapter, and how they can be turned into generic library functions. For example, you might like to write a function named `calcLineStringLength()` that takes a LineString and returns the total length of the LineString's segments, optionally transforming the LineString's coordinates into lat/long values before calling `geod.inv()`. You could then write a `calcPolygonOutlineLength()` function that uses `calcLineStringLength()` to calculate the length of a polygon's outer ring.

- You could then write a `calcPolygonOutlineLength()` function that uses `calcLineStringLength()` to calculate the length of a polygon's outer ring.

Summary

In this chapter, we have looked at various techniques for using OGR, GDAL, Shapely, and `pyproj` within Python programs to solve real-world problems. We have learned:

- How to read from and write to vector-format geo-spatial data in Shapefiles.
- How to read and analyze raster-format geo-spatial data.
- How to change the datum and projection used by a Shapefile.
- That the Well-Known Text (WKT) format can be used to represent geo-spatial features and spatial references in plain text.
- That WKT can be used to transfer geo-spatial data from one Python library to another.
- That WKT can be used to store geo-spatial data in plain text format.
- That you can use the Shapely library to perform various geo-spatial calculations on geometries, including distance calculations, dilation, and intersections.
- That you can use the `pyproj.Proj` class to convert coordinates from one projection and datum to another.
- That you can use the `pyproj.Geod` class to convert from geometry coordinates to distances, and *vice versa*.

Up to now, we have written programs that work directly with Shapefiles and other datasources to load and then process geo-spatial data. In the next chapter, we will look at ways that databases can be used to store and work with geo-spatial data. This is much faster and more scalable than storing geo-spatial data in files that have to be imported each time.