Python Geospatial Analysis Essentials

Python is a highly expressive language that makes it easy to write sophisticated programs. Combining high-quality geospatial data with Python geospatial libraries will give you a powerful toolkit for solving a range of geospatial programming tasks.

The book begins with an introduction to geospatial analysis and programming and explains the ideas behind geospatial data. You will explore Python libraries for building your own geospatial applications. You will learn to create a geospatial database for your application using PostGIS and the psycopg2 library, and see how the Mapnik library can be used to create attractive and useful maps.

Finally, you will learn to use the Shapely and NetworkX libraries to create, analyze, and manipulate complex geometric objects, before implementing a system to match GPS recordings against a database of roads to produce a heatmap of the most frequently used roads.

Who this book is written for

If you are an experienced Python developer and wish to get up-to-speed with geospatial programming, then this book is for you. While familiarity with installing third-party Python libraries would be an advantage, no prior knowledge of geospatial programming is required.

What you will learn from this book

- Understand the key geospatial concepts and techniques needed to analyze and work with geospatial data
- Learn how to read and write geospatial data from within your Python code
- Use PostGIS to store spatial data and perform spatial queries
- Use Python libraries to analyze and manipulate geospatial data
- Generate maps based on your spatial data
- Implement complete geospatial analysis systems using Python
- Use the Shapely and NetworkX libraries to solve problems such as distance-area calculations, finding the shortest path between two points, buffering polygons, and much more
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Geospatial Analysis and Techniques'
- A synopsis of the book’s content
- More information on Python Geospatial Analysis Essentials
About the Author

Erik Westra has been a professional software developer for over 25 years now, and he 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, as well as real-time messaging and payment systems. 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.

Erik is also the author of the titles *Python Geospatial Development* and *Building Mapping Applications with QGIS*, both by Packt Publishing.
Preface

There are several powerful Python libraries for reading, processing, analyzing, and viewing geospatial data. There are also a number of websites that provide high-quality geospatial data, which you can use freely in your own projects. This data will often be the basis for your analysis, providing the shapes of countries, the positions of cities, the outlines of roads, and so on. Using this data in conjunction with the available geospatial libraries gives you a powerful toolkit for performing your own geospatial analysis using Python.

What this book covers

Chapter 1, Geospatial Analysis and Techniques, walks the reader through the process of downloading sample geospatial data, before writing a simple Python program to read and analyze that sample data.

Chapter 2, Geospatial Data, focuses on the data used for geospatial analysis: how to obtain it, why good data is important, the different formats that geospatial data can come in, and how to generate your own spatial datasets.

Chapter 3, Spatial Databases, provides a brief introduction to creating geospatial databases, how to store data in a spatially-enabled database, and how to perform efficient queries against that data.

Chapter 4, Creating Maps, looks at how to use the Mapnik library to produce great-looking maps.

Chapter 5, Analyzing Geospatial Data, guides the reader through the process of writing spatial analysis programs using Python. Based on the datasets downloaded in Chapter 2, Geospatial Data, and using the major Python libraries for geospatial analysis, this chapter uses a recipe-like format to solve a range of typical spatial analysis problems.
Chapter 6, Building a Complete Geospatial Analysis System, uses all the various libraries and techniques covered in the earlier chapters to build a complete geospatial analysis system.
In this introductory chapter, we will start our exploration of geospatial analysis by learning about the types of tasks you will typically be performing, and then look at spatial data and the Python libraries you can use to work with it. We will finish by writing an example program in Python to analyze some geospatial data.

As you work through this chapter, you will:

- Become familiar with the types of problems that geospatial analysis will help to solve
- Understand the various types of geospatial data and some of the important concepts related to location-based data
- Set up your computer to use the third-party libraries you need to start analyzing geospatial data using Python
- Obtain some basic geospatial data to get started
- Learn how to use the GDAL/OGR library to read through a shapefile and extract each feature's attributes and geometry
- Learn how to use Shapely to manipulate and analyze geospatial data
- Write a simple but complete program to identify neighboring countries

Let's start by looking at the types of problems and tasks typically solved using geospatial analysis.
About geospatial analysis

Geospatial analysis is the process of reading, manipulating, and summarizing geospatial data to yield useful and interesting results. A lot of the time, you will be answering questions like the following:

- What is the shortest drivable distance between Sausalito and Palm Springs?
- What is the total length of the border between France and Belgium?
- What is the area of each National Park in New Zealand that borders the ocean?

The answer to these sorts of questions will typically be a number or a list of numbers. Other types of geospatial analysis will involve calculating new sets of geospatial data based on existing data. For example:

- Calculate an elevation profile for USA Route 66 from Los Angeles, CA, to Albuquerque, NM.
- Show me the portion of Brazil north of the equator.
- Highlight the area of Rarotonga likely to be flooded if the ocean rose by 2 meters.

In these cases, you will be generating a new set of geospatial data, which you would typically then display in a chart or on a map.

To perform this sort of analysis, you will need two things: appropriate geospatial analysis tools and suitable geospatial data.

We are going to perform some simple geospatial analysis shortly. Before we do, though, let's take a closer look at the concept of geospatial data.

Understanding geospatial data

Geospatial data is data that positions things on the Earth's surface. This is a deliberately vague definition that encompasses both the idea of location and shape. For example, a database of car accidents may include the latitude and longitude coordinates identifying where each accident occurred, and a file of county outlines would include both the position and shape of each county. Similarly, a GPS recording of a journey would include the position of the traveler over time, tracing out the path they took on their travels.
It is important to realize that geospatial data includes more than just the geospatial information itself. For example, the following outlines are not particularly useful by themselves:

Once you add appropriate metadata, however, these outlines make a lot more sense:

Geospatial data, therefore, includes both spatial information (locations and shapes) and non-spatial information (metadata) about each item being described.

Spatial information is usually represented as a series of coordinates, for example:

```
location = (-38.136734, 176.252300)
outline = ((-61.686,17.024),(-61.738,16.989),(-61.829,16.996) ...)
```

These numbers won't mean much to you directly, but once you plot these series of coordinates onto a map, the data suddenly becomes comprehensible:
There are two fundamental types of geospatial data:

- **Raster data**: This is geospatial data that divides the world up into cells and associates values with each cell. This is very similar to the way that bitmapped images divide an image up into pixels and associate a color with each pixel; for example:

![Raster Data Example](image)

The value of each cell might represent the color to use when drawing the raster data on a map—this is often done to provide a raster basemap on which other data is drawn—or it might represent other information such as elevation, moisture levels, or soil type.

- **Vector data**: This is geospatial data that consists of a list of features. For example, a shapefile containing countries would have one feature for each country. For each feature, the geospatial dataset will have a geometry, which is the shape associated with that feature, and any number of attributes containing the metadata for that feature.

A feature's geometry is just a geometric shape that is positioned on the surface of the earth. This geometric shape is made up of **points**, **lines** (sometimes referred to as **LineStrings**), and **polygons**, or some combination of these three fundamental types:

![Vector Data Example](image)

The typical raster data formats you might encounter include:

- GeoTIFF files, which are basically just TIFF format image files with georeferencing information added to position the image accurately on the earth's surface.
- USGS **.dem** files, which hold a **Digital Elevation Model (DEM)** in a simple ASCII data format.
- **.png**, **.bmp**, and **.jpeg** format image files, with associated georeferencing files to position the images on the surface of the earth.
For vector-format data, you may typically encounter the following formats:

- **Shapefile**: This is an extremely common file format used to store and share geospatial data.
- **WKT (Well-Known Text)**: This is a text-based format often used to convert geometries from one library or data source to another. This is also the format commonly used when retrieving features from a database.
- **WKB (Well-Known Binary)**: This is the binary equivalent of the WKT format, storing geometries as raw binary data rather than text.
- **GML (Geometry Markup Language)**: This is an industry-standard format based on XML, and is often used when communicating with web services.
- **KML (Keyhole Markup Language)**: This is another XML-based format popularized by Google.
- **GeoJSON**: This is a version of JSON designed to store and transmit geometry data.

Because your analysis can only be as good as the data you are analyzing, obtaining and using good-quality geospatial data is critical. Indeed, one of the big challenges in performing geospatial analysis is to get the right data for the job. Fortunately, there are several websites which provide free good-quality geospatial data. But if you're looking for a more obscure set of data, you may have trouble finding it. Of course, you do always have the choice of creating your own data from scratch, though this is an extremely time-consuming process.

We will return to the topic of geospatial data in Chapter 2, Geospatial Data, where we will examine what makes good geospatial data and how to obtain it.

**Setting up your Python installation**

To start analyzing geospatial data using Python, we are going to make use of two freely available third-party libraries:

- **GDAL**: The Geospatial Data Abstraction Library makes it easy for you to read and write geospatial data in both vector and raster format.
- **Shapely**: As the name suggests, this is a wonderful library that enables you to perform various calculations on geometric shapes. It also allows you to manipulate shapes, for example, by joining shapes together or by splitting them up into their component pieces.

Let's go ahead and get these two libraries installed into your Python setup so we can start using them right away.
Installing GDAL

GDAL, or more accurately the GDAL/OGR library, is a project by the Open Source Geospatial Foundation to provide libraries to read and write geospatial data in a variety of formats. Historically, the name GDAL referred to the library to read and write raster-format data, while OGR referred to the library to access vector-format data. The two libraries have now merged, though the names are still used in the class and function names, so it is important to understand the difference between the two.

A default installation of GDAL/OGR allows you to read raster geospatial data in 100 different formats, and write raster data in 71 different formats. For vector data, GDAL/OGR allows you read data in 42 different formats, and write in 39 different formats. This makes GDAL/OGR an extremely useful tool to access and work with geospatial data.

GDAL/OGR is a C++ library with various bindings to allow you to access it from other languages. After installing it on your computer, you typically use the Python bindings to access the library using your Python interpreter. The following diagram illustrates how these various pieces all fit together:

Let's go ahead and install the GDAL/OGR library now. The main website of GDAL (and OGR) can be found at http://gdal.org.

How you install it depends on which operating system your computer is using:

- For MS Windows machines, you can install GDAL/OGR using the FWTools installer, which can be downloaded from http://fwtools.maptools.org.
Alternatively, you can install GDAL/OGR and Shapely using the OSGeo installer, which can be found at http://trac.osgeo.org/osgeo4w.

- For Mac OS X, you can download the complete installer for GDAL and OGR from http://www.kyngchaos.com/software/frameworks.
- For Linux, you can download the source code to GDAL/OGR from the main GDAL site, and follow the instructions on the site to build it from source. You may also need to install the Python bindings for GDAL and OGR.

Once you have installed it, you can check that it's working by firing up your Python interpreter and typing `import osgeo.gdal` and then `import osgeo.ogr`. If the Python command prompt reappears each time without an error message, then GDAL and OGR were successfully installed and you're all ready to go:

```
>>> import osgeo.gdal
>>> import osgeo.ogr
>>> 
```

### Installing Shapely

Shapely is a geometry manipulation and analysis library. It is based on the Geometry Engine, Open Source (GEOS) library, which implements a wide range of geospatial data manipulations in C++. Shapely provides a Pythonic interface to GEOS, making it easy to use these manipulations directly within your Python programs. The following illustration shows the relationship between your Python code, the Python interpreter, Shapely, and the GEOS library:
The main website for Shapely can be found at http://pypi.python.org/pypi/Shapely.

The website has everything you need, including complete documentation on how to use the library. Note that to install Shapely, you need to download both the Shapely Python package and the underlying GEOS library. The website for the GEOS library can be found at http://trac.osgeo.org/geos.

How you go about installing Shapely depends on which operating system your computer is using:

- For MS Windows, you should use one of the prebuilt installers available on the Shapely website. These installers include their own copy of GEOS, so there is nothing else to install.
- For Mac OS X, you should use the prebuilt GEOS framework available at http://www.kyngchaos.com/software/frameworks.

Note that if you install the GDAL Complete package from the preceding website, you will already have GEOS installed on your computer.

Once GEOS has been installed, you can install Shapely using pip, the Python package manager:

```
pip install shapely
```

If you don't have pip installed on your computer, you can install it by following the instructions at https://pip.pypa.io/en/latest/installing.html.

- For Linux machines, you can either download the source code from the GEOS website and compile it yourself, or install a suitable RPM or APT package which includes GEOS. Once this has been done, you can use pip install shapely to install the Shapely library itself.

Once you have installed it, you can check that the Shapely library is working by running the Python command prompt and typing the following command:

```
>>> import shapely.geos

>>> import shapely.geos
```

If you get the Python command prompt again without any errors, as in the preceding example, then Shapely has been installed successfully and you're all set to go.
Obtaining some geospatial data

For this chapter, we will use a simple but still very useful geospatial data file called World Borders Dataset. This dataset consists of a single shapefile where each feature within the shapefile represents a country. For each country, the associated geometry object represents the country's outline. Additional attributes contain metadata such as the name of the country, its ISO 3166-1 code, the total land area, its population, and its UN regional classification.


Scroll down to the Downloads section and click on the file to download. Make sure you download the full version and not the simplified one—the file you want will be called TM_WORLD_BORDERS-0.3.zip.

Note that the shapefile comes in the form of a ZIP archive. This is because a shapefile consists of multiple files, and it is easier to distribute them if they are stored in a ZIP archive. After downloading the file, double-click on the ZIP archive to decompress it. You will end up with a directory named TM_WORLD_BORDERS-0.3. Inside this directory should be the following files:

The following table explains these various files and what information they contain:

<table>
<thead>
<tr>
<th>Filename</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Readme.txt</td>
<td>This is your typical README file, containing useful information about the shapefile.</td>
</tr>
<tr>
<td>TM_WORLD_BORDERS-0.3.shp</td>
<td>This file contains the geometry data for each feature.</td>
</tr>
<tr>
<td>TM_WORLD_BORDERS-0.3.shx</td>
<td>This is an index into the .shp file, making it possible to quickly access the geometry for a given feature.</td>
</tr>
<tr>
<td>TM_WORLD_BORDERS-0.3.dbf</td>
<td>This is a database file holding the various attributes for each feature.</td>
</tr>
<tr>
<td>TM_WORLD_BORDERS-0.3.prj</td>
<td>This file describes the coordinate system and projection used by the data, as a plain text file.</td>
</tr>
</tbody>
</table>
Place this directory somewhere convenient. We will be using this dataset extensively throughout this book, so you may want to keep a backup copy somewhere.

**Unlocking the shapefile**

At last, we are ready to start working with some geospatial data. Open up a command line or terminal window and `cd` into the `TM_WORLD_BORDERS-0.3` directory you unzipped earlier. Then type `python` to fire up your Python interpreter.

We're going to start by loading the OGR library we installed earlier:

```python
>>> import osgeo.ogr
```

We next want to open the shapefile using OGR:

```python
>>> shapefile = osgeo.ogr.Open("TM_WORLD_BORDERS-0.3.shp")
```

After executing this statement, the `shapefile` variable will hold an `osgeo.ogr.Datasource` object representing the geospatial data source we have opened. OGR data sources can support multiple layers of information, even though a shapefile has only a single layer. For this reason, we next need to extract the (one and only) layer from the shapefile:

```python
>>> layer = shapefile.GetLayer(0)
```

Let's iterate through the various features within the shapefile, processing each feature in turn. We can do this using the following:

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

The `feature` object, an instance of `osgeo.ogr.Feature`, allows us to access the geometry associated with the feature, along with the feature's attributes. According to the `README.txt` file, the country's name is stored in an attribute called `NAME`. Let's extract that name now:

```python
>>> feature_name = feature.GetField("NAME")
```

Notice that the attribute is in uppercase. Shapefile attributes are case sensitive, so you have to use the exact capitalization to get the right attribute. Using `feature.getField("name")` would generate an error.
To get a reference to the feature's geometry object, we use the `GetGeometryRef()` method:

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

We can do all sorts of things with geometries, but for now, let's just see what type of geometry we've got. We can do this using the `GetGeometryName()` method:

```python
>>> geometry_type = geometry.GetGeometryName()
```

Finally, let's print out the information we have extracted for this feature:

```python
>>> print i, feature_name, geometry_type
```

Here is the complete mini-program we've written to unlock the contents of the shapefile:

```python
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)
    feature_name = feature.GetField("NAME")
    geometry = feature.GetGeometryRef()
    geometry_type = geometry.GetGeometryName()
    print i, feature_name, geometry_type
```

If you press Return a second time to close off the for loop, your program will run, displaying useful information about each country extracted from the shapefile:

```
0 Antigua and Barbuda MULTIPOLYGON
1 Algeria POLYGON
2 Azerbaijan MULTIPOLYGON
3 Albania POLYGON
4 Armenia MULTIPOLYGON
5 Angola MULTIPOLYGON
6 American Samoa MULTIPOLYGON
7 Argentina MULTIPOLYGON
8 Australia MULTIPOLYGON
9 Bahrain MULTIPOLYGON
...```
Notice that the geometry associated with some countries is a polygon, while for other countries the geometry is a multipolygon. As the name suggests, a multipolygon is simply a collection of polygons. Because the geometry represents the outline of each country, a polygon is used where the country's outline can be represented by a single shape, while a multipolygon is used when the outline has multiple parts. This most commonly happens when a country is made up of multiple islands. For example:

As you can see, Algeria is represented by a polygon, while Australia with its outlying islands would be a multipolygon.

**Analyzing the data**

In the previous section, we obtained an `osgeo.ogr.Geometry` object representing each country's outline. While there are a number of things we can do with this geometry object directly, in this case we'll take the outline and copy it into Shapely so that we can take advantage of Shapely's geospatial analysis capabilities. To do this, we have to export the geometry object out of OGR and import it as a Shapely object. For this, we'll use the WKT format. Still in the Python interpreter, let's grab a single feature's geometry and convert it into a Shapely object:

```python
>>> import shapely.wkt
>>> feature = layer.GetFeature(0)
>>> geometry = feature.GetGeometryRef()
>>> wkt = geometry.ExportToWkt()
>>> outline = shapely.wkt.loads(wkt)
```
Because we loaded feature number 0, we retrieved the outline for Antigua and Barbuda, which would look like the following if we displayed it on a map:

![Outline of Antigua and Barbuda](image)

The `outline` variable holds the outline of this country in the form of a Shapely `MultiPolygon` object. We can now use this object to analyze the geometry. Here are a few useful things we can do with a Shapely geometry:

- We can calculate the **centroid**, which is the center-most point in the geometry.
- We can calculate the **bounding box** for the geometry. This is a rectangle defining the northern, southern, eastern, and western edges of the polygon.
- We can calculate the **intersection** between two geometries.
- We can calculate the **difference** between two geometries.

We could also calculate values such as the length and area of each polygon. However, because the World Borders Dataset uses what are called *unprojected coordinates*, the resulting length and area values would be measured in degrees rather than meters or miles. This means that the calculated lengths and areas wouldn't be very useful. We will look at the nature of map projections in the following chapter and find a way to get around this problem so we can calculate meaningful length and area values for polygons. But that's too complex for us to tackle right now.
Let's display the latitude and longitude for our feature's centroid:

```python
>>> print outline.centroid.x, outline.centroid.y
-61.791127517 17.2801365868
```

Because Shapely doesn't know which coordinate system the polygon is in, it uses the more generic $x$ and $y$ attributes for a point, rather than talking about latitude and longitude values. Remember that latitude corresponds to a position in the north-south direction, which is the $y$ value, while longitude is a position in the east-west direction, which is the $x$ value.

We can also display the outline's bounding box:

```python
>>> print outline.bounds
(-61.891113, 16.989719, -61.666389, 17.724998)
```

In this case, the returned values are the minimum longitude and latitude and the maximum longitude and latitude (that is, $\min_x, \min_y, \max_x, \max_y$).

There's a lot more we can do with Shapely, of course, but this is enough to prove that the Shapely library is working, and that we can read geospatial data from a shapefile and convert it into a Shapely geometry object for analysis.

This is as far as we want to go with using the Python shell directly—the shell is great for quick experiments like this, but it quickly gets tedious having to retype lines (or use the command history) when you make a typo. For anything more serious, you will want to write a Python program. In the final section of this chapter, we'll do exactly that: create a Python program that builds on what we have learned to solve a useful geospatial analysis problem.

**A program to identify neighboring countries**

For our first real geospatial analysis program, we are going to write a Python script that identifies neighboring countries. The basic concept is to extract the polygon or multipolygon for each country and see which other countries each polygon or multipolygon touches. For each country, we will display a list of other countries that border that country.
Let's start by creating the Python script. Create a new file named `borderingCountries.py` and place it in the same directory as the `TM_WORLD_BORDERS-0.3.shp` shapefile you downloaded earlier. Then enter the following into this file:

```python
import osgeo.ogr
import shapely.wkt

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

    countries = {}  # Maps country name to Shapely geometry.

    for i in range(layer.GetFeatureCount()):
        feature = layer.GetFeature(i)
        country = feature.GetField("NAME")
        outline = shapely.wkt.loads(feature.GetGeometryRef().ExportToWkt())
        countries[country] = outline

    print "Loaded %d countries" % len(countries)

if __name__ == "__main__":
    main()
```

So far, this is pretty straightforward. We are using the techniques we learned earlier to read the contents of the shapefile into memory and converting each country's geometry into a Shapely object. The results are stored in the `countries` dictionary. Finally, notice that we've placed the program logic into a function called `main()` — this is good practice as it lets us use a `return` statement to handle errors.

Now run your program just to make sure it works:

```
$ python borderingCountries.py
Loaded 246 countries
```

Our next task is to identify the bordering countries. Our basic logic will be to iterate through each country and then find the other countries that border this one. Here is the relevant code, which you should add to the end of your `main()` function:

```python
for country in sorted(countries.keys()):
    outline = countries[country]

    for other_country in sorted(countries.keys()):
```
if country == other_country: continue

other_outline = countries[other_country]

if outline.touches(other_outline):
    print "%s borders %s" % (country, other_country)

As you can see, we use the touches() method to check if the two countries' geometries are touching.

Running this program will now show you the countries that border each other:

$ python borderingCountries.py
Loaded 246 countries
Afghanistan borders Tajikistan
Afghanistan borders Uzbekistan
Albania borders Montenegro
Albania borders Serbia
Albania borders The former Yugoslav Republic of Macedonia
Algeria borders Libyan Arab Jamahiriya
Algeria borders Mali
Algeria borders Morocco
Algeria borders Niger
Algeria borders Western Sahara
Angola borders Democratic Republic of the Congo
Argentina borders Bolivia
...

Congratulations! You have written a simple Python program to analyze country outlines. Of course, there is a lot that could be done to improve and extend this program. For example:

- You could add command-line arguments to let the user specify the name of the shapefile and which attribute to use to display the country name.
- You could add error checking to handle invalid and non-existent shapefiles.
- You could add error checking to handle invalid geometries.
- You could use a spatial database to speed up the process. The program currently takes about a minute to complete, but using a spatial database would speed that up dramatically. If you are dealing with a large amount of spatial data, properly indexed databases are absolutely critical or your program might take weeks to run.

We will look at all these things later in the book.
Summary

In this chapter, we started our exploration of geospatial analysis by looking at the types of problems you would typically have to solve and the types of data that you will be working with. We discovered and installed two major Python libraries to work with geospatial data: GDAL/OGR to read (and write) data, and Shapely to perform geospatial analysis and manipulation. We then downloaded a simple but useful shapefile containing country data, and learned how to use the OGR library to read the contents of that shapefile.

Next, we saw how to convert an OGR geometry object into a Shapely geometry, and then used the Shapely library to analyze and manipulate that geometry. Finally, we created a simple Python program that combines everything we have learned, loading country data into memory and then using Shapely to find countries which border each other.

In the next chapter, we will delve deeper into the topic of geospatial data, learning more about geospatial data types and concepts, as well as exploring some of the major sources of freely available geospatial data. We will also learn why it is important to have good data to work with—and what happens if you don't.
Where to buy this book
You can buy Python Geospatial Analysis Essentials from the Packt Publishing website. Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.
Click here for ordering and shipping details.