Geospatial Development By Example with Python

Python is a powerful and easy-to-use programming language with a huge community and extensive availability of free and open source packages and libraries dedicated to data processing and analysis. With the ever-growing amount of geographic data available, Python provides the tools and means for efficient and automated data processing, yielding results in a reasonable amount of time.

From Python programming best practices to the advanced use of analysis packages, this book teaches you how to write applications that will perform complex geoprocessing tasks that can be replicated and reused. Much more than simple scripts, you will write functions to import data, create Python classes that represent your features, and learn how to combine and filter them.

Who this book is written for
Geospatial Development By Example with Python is intended for beginner or advanced Python developers who want to work with geographic data. The book is suitable for professional developers who are new to geospatial development, for hobbyists, or for data scientists who want to move into some simple development.

What you will learn from this book
- Prepare a development environment with all the tools needed for geoprocessing with Python
- Import point data and structure an application using Python's resources
- Combine point data from multiple sources, creating intuitive and functional representations of geographic objects
- Filter data by coordinates or attributes easily using pure Python
- Download, transform, and use remote sensing data in your maps
- Make calculations to extract information from raster data and show the results on beautiful maps
- Handle massive amounts of data with advanced processing techniques
- Process huge satellite images in an efficient way
- Optimize geoprocessing times with parallel processing

Build your first interactive map and build location-aware applications using cutting-edge examples in Python

Pablo Carreira

$ 49.99 US
£ 31.99 UK

Prices do not include local sales tax or VAT
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Preparing the Work Environment title'
- A synopsis of the book’s content
- More information on Geospatial Development By Example with Python
About the Author

**Pablo Carreira** is a Python programmer and a full stack developer living in São Paulo state, Brazil. He is now the lead developer of an advanced web platform for precision agriculture and actively uses Python as a backend solution for efficient geoprocessing.

Born in 1980, Brazil, Pablo graduated as an agronomical engineer. Being a programming enthusiast and self-taught since childhood, he learned programming as a hobby and later honored his techniques in order to solve work tasks.

Having 8 years of professional experience in geoprocessing, he uses Python along with geographic information systems in order to automate processes and solve problems related to precision agriculture, environmental analysis, and land division.
Preface

From Python programming good practices to the advanced use of analysis packages, this book teaches how to write applications that will perform complex geoprocessing tasks that can be replicated and reused. The book contains three sample applications. Chapter 1 shows how to prepare a development environment. From Chapter 2 to Chapter 4, the reader goes deep into Python functionality using classes, inheritance, and other resources in order to read, manipulate, combine, and search information in vector data. Chapter 5 to Chapter 7 presents techniques to render beautiful maps, and handle and analyze raster data. In the final three chapters, the book approaches code optimization and presents solutions to handle large datasets common in geoprocessing tasks. All the examples are modular and can be rearranged to achieve countless different results. During the book, the code is deduced step by step until it reaches the final form. The reader is led to edit, change, and improve the code, experimenting with different solutions and organizations, subtly learning the mental process of the development of a geoprocessing application.

What this book covers

Chapter 1, Preparing the Work Environment, shows the processes of installing all the libraries that you will need to go through the examples in the book, as well how to set up an integrated development environment (IDE) that will help organize the code and avoid mistakes. Finally, it will present the first contact with one of the geospatial libraries.

Chapter 2, The Geocaching App, will go through the important steps in geoprocessing applications, such as opening files, reading data, and preparing it for analysis with the tools at hand. Going through these steps, the user will learn how to organize and use the resources provided by the language to write consistent applications.
Chapter 3, Combining Multiple Data Sources, will cover the process of combining sources, and how to use Python classes to create your own representation of geospatial data. Geographic data tends to be heterogeneous, so writing programs that are able to combine multiple sources of data is a fundamental topic in geoprocessing.

Chapter 4, Improving the App Search Capabilities, will add new functionalities to the application. Users will write a code capable of filtering features by geographic boundaries and by any field in the data. In the process, they will see how to work with polygons and how the relations between geometries can be analyzed in a geoprocessing application.

Chapter 5, Making Maps, starts a new application that will be able to produce nice maps from the vectors of the data. Mapnik, one of the most used mapping packages, will be used. The user will understand how it works and how to adapt it to consume the data presented in the previous chapters.

Chapter 6, Working with Remote Sensing Images, will present a process of deduction that will result in a versatile and powerful software structure able to combine, crop, and adjust the values of images to prepare them for presentation.

Chapter 7, Extract Information from Raster Data, will approach the process of extracting information from raster data, which can be analyzed in order to produce valuable information. Going beyond simple numerical values, it will show how to display this information on beautiful color maps.

Chapter 8, Data Miner App, will show how to use databases and how to import data into it in order to minimize processing time and allow huge datasets to be processed. Geospatial data tends to be extensive and its processing demands a lot of computer power. To make the code more efficient, the reader will learn techniques of code profiling and optimization.

Chapter 9, Processing Big Images, will show how to process big satellite images. It will focus on how to perform sustainable image processing and how to open and make calculations with many big images while keeping the memory consumption low with efficient code.

Chapter 10, Parallel Processing, will teach the reader how to use the full available computer power. In order to speed up tasks, it will show how to distribute them among processor cores for parallel processing.
Preparing the Work Environment

Working with a programming language as a tool for geoprocessing provides the opportunity to construct a personalized application that can more optimally perform the task required by the user. This means that repetitive tasks can be automated, file inputs and outputs can be customized, and processes can be tuned to perform exactly what you want to be done.

Python is a powerful programming language that is gaining special attention as a tool for geoprocessing and scientific analysis. A number of factors may have contributed to its popularization, and three among them are worth mentioning: it's a scripting language, it's flexible and easy to learn, and it has a wide range of libraries available as open source software.

The number of available libraries and packages allow users to spend less time in programming basic functionalities and more in building processes and workflows to reach their goals.

In this first chapter, we will go through the process of installing all the libraries that you will need to go through the examples; it's likely that these same libraries will also satisfy most of your needs in real-world applications. Then, we will set up an Integrated Development Environment (IDE) that will help organize code and avoid mistakes. Finally, we will write a sample program with one of the libraries. Therefore, here are the topics that will be covered:

- Installing Python and the packages that you need for the examples in this book
- Learning the basics of the packages that you will use
Preparing the Work Environment

- Installing an IDE to write and organize your code
- Creating a project for this book
- Writing your first code

Installing Python

For this book, we suggest using Python 2.7; this version of Python is fully compatible with the libraries and packages that we will use in the examples and also has precompiled binary files available on the Internet for Windows users. We will keep all the examples as compatible as possible with Python 3.4 so that it would be easy to port future upgrades.

Windows users may find compatibility problems with the 64-bit packages, so we recommend the 32-bit version of Python for them.

For Linux users, we will show the installation procedures for Ubuntu Linux distribution and use package managers, so you don't need to worry about versions and requirements; the package managers will do this for you.

The libraries that you will install are written in Python and other languages, the most common being C and C++. These libraries can abstract classes, methods, and functions to Python objects or have an extra layer that makes the connection; when this happens, we say that the library has Python bindings.

Windows

Here are the steps to perform for the installation of Python on Windows:

1. Go to https://www.python.org/downloads/windows/ and click on Download the latest Python 2.7 release for Windows.
2. On the next page, roll down, and you will find a list of files; make sure that you download Windows x86 MSI installer.
3. After the file is downloaded, open the installer by clicking on the file and following the instructions. Proceed with the default options by clicking on the Next button.

Ubuntu Linux

Ubuntu already comes with Python installed, so there is no need to install it. If for any reason, it's not available, you can install it with the following command:

    sudo apt-get install python
Python 2.7.9 comes with Pip, but if you use an older version, you need to install Pip with the following command:

```
sudo apt-get install python-pip
```

**Python packages and package manager**

A Python package is a directory containing one or more Python files (that is, modules) plus one `__init__.py` file (this can be just an empty file). This file tells Python Interpreter that the directory is a package.

When writing Python code, we can import packages and modules and use them in our code. The Python community does this a lot; many packages use other packages and so on, forming an intricate network of requirements and dependencies.

In order to facilitate the installation of packages and all the requirements for it to run, Python has a package manager called pip.

Pip looks for packages in a central repository (or on a user-defined place), downloads it, then downloads its dependencies, and installs them. Some packages also use libraries in other languages, such as C. In these cases, these libraries need to be compiled during the installation. Ubuntu users don't have problem with this because many compilers are already installed on the system, but this won't work on Windows by default.

**The repository of Python packages for Windows**

Python makes it easy to install libraries and packages through pip. However, since Windows doesn't include any compiler by default, the installation of packages that needs the compilation of libraries fails. Instead of going through the process of installing a compiler, which is out of this book's scope, we can get the packages ready to use.

These packages come prebuilt for various types of system and don't need a compilation of its libraries. This type of package is called a wheel.

---

Christoph Gohlke did a favor to all of us by building these packages and making them available for download at http://www.lfd.uci.edu/~gohlke/pythonlibs/.

---
Preparing the Work Environment

Installing packages and required software
In this topic, we will go through the installation process of every package used in the book.

OpenCV
OpenCV is an optimized C/C++ library intended for video and image processing with hundreds of functions ranging from simple image resizing to object recognition, face detection, and so on. OpenCV is a big library, and we will use its capabilities of reading, transforming, and writing images. It’s a good choice because its development is active, and it has a large user community and very good documentation.

Windows
Here is the installation procedure for Windows:

2. Press Ctrl + F to open the search dialog of your browser and then search for OpenCV.
3. You will find a list of files; choose opencv_python-2.4.11-cp27-none-win32.whl or any OpenCV version that contains cp27 and win32. This means that this is the 32-bit version for Python 2.7.
4. Save the downloaded file to a known location.
5. Open Windows Command Prompt and run the following command:
c:\Python27\scripts\pip install path_to_the_file_you_downloaded.whl
6. You should see an output telling you that the installation was successful, as follows:
Processing c:\downloads\opencv_python-2.4.12-cp27-none-win32.whl
Installing collected packages: opencv-python
Successfully installed opencv-python-2.4.12

You can drag and drop a file into the command prompt to enter its full path.
**Ubuntu Linux**

Here is the installation process for Ubuntu Linux:

1. Open a new terminal with Ctrl + T.
2. Then, enter the following command:
   `sudo apt-get install python-opencv`

**Installing NumPy**

NumPy is a package for scientific computing with Python. It handles multidimensional arrays of operations in a very efficient way. NumPy is required by OpenCV to run and will be used by many raster operations that we will perform in the examples. NumPy is also an efficient data container and will be our tool to calculate massive image data.

**Windows**

Repeat the same procedure as you did to install OpenCV; however, this time, search for NumPy and choose a file named `numpy-1.9.2+mkl-cp27-none-win32.whl`.

**Ubuntu Linux**

NumPy is automatically installed as a dependency of OpenCV on Ubuntu, but if you want to install it without OpenCV, follow these steps:

1. Open a new terminal with Ctrl + T.
2. Then, enter the following command:
   `sudo pip install numpy`

**Installing GDAL and OGR**

GDAL (Geospatial Data Abstraction Library) is composed of two packages that come together: OGR handles geospatial vector file formats, including coordinate system transformations and vector operations. GDAL is the raster part of the library, and in version 1.11, it comes packed with 139 drivers that can read, and some even create rasters. GDAL also comes packed with functions for raster transformations and calculations such as resizing, clipping, reprojecting, and so on.
Preparing the Work Environment

In the following tables, there's an excerpt of the list of GDAL and OGR drivers with the most common formats that you may find:

<table>
<thead>
<tr>
<th>Long format name</th>
<th>Code</th>
<th>Creation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arc/Info ASCII Grid</td>
<td>AAIGrid</td>
<td>Yes</td>
</tr>
<tr>
<td>Arc/Info Export E00 GRID</td>
<td>E00GRID</td>
<td>No</td>
</tr>
<tr>
<td>ENVI .hdr Labelled Raster</td>
<td>ENVI</td>
<td>Yes</td>
</tr>
<tr>
<td>Generic Binary (.hdr Labelled)</td>
<td>GENBIN</td>
<td>No</td>
</tr>
<tr>
<td>Oracle Spatial GeoRaster</td>
<td>GEORASTER</td>
<td>Yes</td>
</tr>
<tr>
<td>GSat File Format</td>
<td>GFF</td>
<td>No</td>
</tr>
<tr>
<td>Graphics Interchange Format (.gif)</td>
<td>GIF</td>
<td>Yes</td>
</tr>
<tr>
<td>GMT Compatible netCDF</td>
<td>GMT</td>
<td>Yes</td>
</tr>
<tr>
<td>GRASS ASCII Grid</td>
<td>GRASSASCIIGrid</td>
<td>No</td>
</tr>
<tr>
<td>Golden Software ASCII Grid</td>
<td>GSAG</td>
<td>Yes</td>
</tr>
<tr>
<td>Golden Software Binary Grid</td>
<td>GSBG</td>
<td>Yes</td>
</tr>
<tr>
<td>Golden Software Surfer 7 Binary Grid</td>
<td>GS7BG</td>
<td>Yes</td>
</tr>
<tr>
<td>TIFF / BigTIFF / GeoTIFF (.tif)</td>
<td>GTiff</td>
<td>Yes</td>
</tr>
<tr>
<td>GXF (Grid eXchange File)</td>
<td>GXF</td>
<td>No</td>
</tr>
<tr>
<td>Erdas Imagine (.img)</td>
<td>HFA</td>
<td>Yes</td>
</tr>
<tr>
<td>JPEG JFIF (.jpg)</td>
<td>JPEG</td>
<td>Yes</td>
</tr>
<tr>
<td>NOAA Polar Orbiter Level 1b Data Set</td>
<td>L1B</td>
<td>No</td>
</tr>
<tr>
<td>(AVHRR)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOAA NGS Geoid Height Grids</td>
<td>NGSGEOID</td>
<td>No</td>
</tr>
<tr>
<td>NITF</td>
<td>NITF</td>
<td>Yes</td>
</tr>
<tr>
<td>NTv2 Datum Grid Shift</td>
<td>NTv2</td>
<td>Yes</td>
</tr>
<tr>
<td>PCI .aux Labelled</td>
<td>PAux</td>
<td>Yes</td>
</tr>
<tr>
<td>PCI Geomatics Database File</td>
<td>PCIDSK</td>
<td>Yes</td>
</tr>
<tr>
<td>PCRaster</td>
<td>PCRaster</td>
<td>Yes</td>
</tr>
<tr>
<td>Geospatial PDF</td>
<td>PDF</td>
<td>Yes</td>
</tr>
<tr>
<td>NASA Planetary Data System</td>
<td>PDS</td>
<td>No</td>
</tr>
<tr>
<td>Portable Network Graphics (.png)</td>
<td>PNG</td>
<td>Yes</td>
</tr>
<tr>
<td>R Object Data Store</td>
<td>R</td>
<td>Yes</td>
</tr>
<tr>
<td>Raster Matrix Format (*.rsw,.mtw)</td>
<td>RMF</td>
<td>Yes</td>
</tr>
<tr>
<td>RadarSat2 XML (product.xml)</td>
<td>RS2</td>
<td>No</td>
</tr>
<tr>
<td>Idrisi Raster</td>
<td>RST</td>
<td>Yes</td>
</tr>
<tr>
<td>SAGA GIS Binary format</td>
<td>SAGA</td>
<td>Yes</td>
</tr>
</tbody>
</table>
### Chapter 1

<table>
<thead>
<tr>
<th>Long format name</th>
<th>Code</th>
<th>Creation</th>
</tr>
</thead>
<tbody>
<tr>
<td>USGS SDTS DEM (*CATD.DDF)</td>
<td>SDTS</td>
<td>No</td>
</tr>
<tr>
<td>SGI Image Format</td>
<td>SGI</td>
<td>Yes</td>
</tr>
<tr>
<td>SRTM HGT Format</td>
<td>SRTM_HGT</td>
<td>Yes</td>
</tr>
<tr>
<td>Terragen Heightfield (.ter)</td>
<td>TERRAGEN</td>
<td>Yes</td>
</tr>
<tr>
<td>USGS ASCII DEM / CDED (.dem)</td>
<td>USGSDEM</td>
<td>Yes</td>
</tr>
<tr>
<td>ASCII Gridded XYZ</td>
<td>XYZ</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The following table describes the OGR drivers:

<table>
<thead>
<tr>
<th>Format name</th>
<th>Code</th>
<th>Creation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arc/Info Binary Coverage</td>
<td>AVCBin</td>
<td>No</td>
</tr>
<tr>
<td>Arc/Info .E00 (ASCII) Coverage</td>
<td>AVCE00</td>
<td>No</td>
</tr>
<tr>
<td>AutoCAD DXF</td>
<td>DXF</td>
<td>Yes</td>
</tr>
<tr>
<td>Comma Separated Value (.csv)</td>
<td>CSV</td>
<td>Yes</td>
</tr>
<tr>
<td>ESRI Shapefile</td>
<td>ESRI Shapefile</td>
<td>Yes</td>
</tr>
<tr>
<td>GeoJSON</td>
<td>GeoJSON</td>
<td>Yes</td>
</tr>
<tr>
<td>Géoconcept Export</td>
<td>Geoconcept</td>
<td>Yes</td>
</tr>
<tr>
<td>GeoRSS</td>
<td>GeoRSS</td>
<td>Yes</td>
</tr>
<tr>
<td>GML</td>
<td>GML</td>
<td>Yes</td>
</tr>
<tr>
<td>GMT</td>
<td>GMT</td>
<td>Yes</td>
</tr>
<tr>
<td>GPSBabel</td>
<td>GPSBabel</td>
<td>Yes</td>
</tr>
<tr>
<td>GPX</td>
<td>GPX</td>
<td>Yes</td>
</tr>
<tr>
<td>GPSTrackMaker (.gtm, .gtz)</td>
<td>GPSTrackMaker</td>
<td>Yes</td>
</tr>
<tr>
<td>Hydrographic Transfer Format</td>
<td>HTF</td>
<td>No</td>
</tr>
<tr>
<td>Idrisi Vector (.VCT)</td>
<td>Idrisi</td>
<td>No</td>
</tr>
<tr>
<td>KML</td>
<td>KML</td>
<td>Yes</td>
</tr>
<tr>
<td>Mapinfo File</td>
<td>MapInfo File</td>
<td>Yes</td>
</tr>
<tr>
<td>Microstation DGN</td>
<td>DGN</td>
<td>Yes</td>
</tr>
<tr>
<td>OpenAir</td>
<td>OpenAir</td>
<td>No</td>
</tr>
<tr>
<td>ESRI FileGDB</td>
<td>OpenFileGDB</td>
<td>No</td>
</tr>
<tr>
<td>PCI Geomatics Database File</td>
<td>PCIDSK</td>
<td>Yes</td>
</tr>
<tr>
<td>Geospatial PDF</td>
<td>PDF</td>
<td>Yes</td>
</tr>
<tr>
<td>PDS</td>
<td>PDS</td>
<td>No</td>
</tr>
<tr>
<td>PostgreSQL SQL dump</td>
<td>PGDump</td>
<td>Yes</td>
</tr>
<tr>
<td>U.S. Census TIGER/Line</td>
<td>TIGER</td>
<td>No</td>
</tr>
</tbody>
</table>
Preparing the Work Environment

You can find the full GDAL and OGR API documentation and the complete list of drivers at http://gdal.org/python/.

Windows
Again, we will use a wheel for the installation. Repeat the same procedure as before:

2. Now, search for GDAL and download the file named GDAL-1.11.3-cp27-none-win32.whl.
3. Finally, install it with pip, as we did before.

Ubuntu Linux
Perform the following steps:

1. Go to the terminal or open a new one.
2. Then, enter the following command:
   sudo apt-get install python-gdal

Installing Mapnik
Mapnik is a map rendering package. It is a free toolkit to develop mapping applications. It produces high-quality maps and is used on many applications, including OpenStreetMaps.

Windows
Mapnik isn't available for installation as other libraries are. Instead, you need to go to http://mapnik.org/ and follow the download link:

1. Download the Windows 32-bit package of Mapnik 2.2.
2. Extract the mapnik-v2.2.0 to C:\ folder.
3. Then, rename the extracted folder c:\mapnik.
4. Now, add Mapnik to your PATH.
5. Open Control Panel and go to System.
6. Click on the Advanced System Settings link in the left-hand side column.
7. In the System Properties window, click on the Advanced tab.
8. Next, click on the Environment Variables button.
9. In the System variables section, highlight the PATH variable and click on Edit. Add the following paths to the end of the list, each separated with a semicolon, as follows:
   c:\mapnik\bin;c:\mapnik\lib
10. Now, click on the New button; then, set the variable name to PYTHONPATH and value to c:\mapnik\python\2.7\site-packages.

Ubuntu Linux
For this, perform the following:

1. Go to the terminal or open a new one.
2. Then, enter the following command:
   sudo apt-get install mapnik

Installing Shapely
Shapely is a package for the manipulation and analysis of two dimensional geometries. It can perform operations such as union and subtraction of geometries. It also can perform tests and comparisons, such as when a geometry intersects other geometries.

Windows
Here's what you need to do:

1. As before, download the prebuilt wheel; this time, look for a file named Shapely-1.5.13-cp27-none-win32.whl.
2. Then, install it with pip.
Preparing the Work Environment

**Ubuntu Linux**
Here are the steps you need to perform:

1. Go to the terminal or open a new one with Ctrl + T.
2. Enter the following command:
   ```bash
   sudo pip install shapely
   ```

**Installing other packages directly from pip**
Some packages do not require compilation steps. For Windows users, these are easier to install because they can be obtained and installed directly with pip with a single command.

**Windows**
You need to simply type the following command in your Command Prompt:
```bash
C:\Python27\scripts\pip install django tabulate requests xmltodict psycopg2
```

**Ubuntu Linux**
In the terminal, type the following command:
```bash
sudo pip install django tabulate requests xmltodict psycopg2
```
For each package, you should see the progress of the installation, similar to the following:

```
Collecting django
  Downloading Django-1.9-py2.py3-none-any.whl (6.6MB)
    100% |################################| 6.6MB 43kB/s
Installing collected packages: django
Successfully installed django-1.9
```
Installing an IDE

IDEs are fancy text editors with tools and inspections regarding programming languages. You can surely use any text editor or IDE of your preference; none of the tasks in this book depends on the IDE, but an IDE will facilitate our work a lot because the suggested configuration will help you avoid mistakes and save time on typing, running, and debugging your code. The IDE checks the code for you and detects underlying errors; it even guesses what you are typing and completes the statements for you, runs the code with a simple command, and if there are exceptions, it provides links to the place where the exception occurred. For Windows or Linux, go to http://www.jetbrains.com/pycharm/ and click on the big orange button Get Pycharm Now. On the next page, select the free community edition.

Windows

Here are the steps you need to perform:

1. After the download finishes, open the downloaded file; the Setup Wizard will pop up.
2. Click on Next, and in the installation options, check both of the boxes: Create Desktop shortcut and Create associations.
3. Click on Next and continue the installation.

Linux

Perform the following steps:

1. Unpack the downloaded file in a directory.
2. To open PyCharm, run pycharm.sh from the bin subdirectory. You can create a shortcut to it if you wish.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.
Preparing the Work Environment

Creating the book project
Perform the following steps:

1. After installation, open Pycharm, and you will be prompted to create your first project:
2. Click on create new project and then choose `c:\geopy` as your project location. In Linux, you can put the project inside your home folder—for example, `/home/myname/geopy`. Click on **Create** to create the project.
Preparing the Work Environment

3. In Windows, you will receive a security alert; this is Pycharm trying to access the Internet. It's recommended that you allow it so that you can later check for updates or download plugins:

![Windows Security Alert]

4. Finally, you should see the following window on your project workspace. Take some time to explore the menus and buttons, try right-clicking on your project folder to see the options:

![Project Window]
Programming and running your first example

Now that we have all we need installed, we will go through our first example. In this example, we will test the installation and then see a glimpse of OGR's capabilities.

To do this, we will open a vector file containing the boundaries of all the countries in the world and make a list of country names. The objective of this simple example is to present the logic behind OGR objects and functions and give an understanding of how geospatial files are represented. Here's how:

1. First, you need to copy the sample data provided with the book to your project folder. You can do this by dragging and dropping the data folder into the geopy folder. Make sure that the data folder is named data and that it's inside the geopy folder.

2. Now, create a new directory for this chapter code, inside PyCharm. With your geopy project opened, right-click on the project folder and select New | Directory. Name it Chapter1.

3. Create a new Python file. To do this, right-click on the Chapter1 folder and select New | Python File. Name it world_borders.py, and PyCharm will automatically open the file for editing.

4. Type the following code in this file:
   ```python
   import ogr
   # Open the shapefile and get the first layer.
   datasource = ogr.Open("../data/world_borders_simple.shp")
   layer = datasource.GetLayerByIndex(0)
   print("Number of features: {}".format(layer.GetFeatureCount()))
   ```

5. Now, run the code; in the menu bar, navigate to Run | Run, and in the dialog, choose world_borders. An output console will open at the bottom of the screen, and if everything goes fine, you should see this output:
   ```none
   C:\Python27\python.exe C:/geopy/world_borders.py
   Number of features: 246
   Process finished with exit code 0
   ```

Congratulations! You successfully opened a Shapefile and counted the number of features inside it. Now, let's understand what this code does.

The first line imports the ogr package. From this point on, all the functions are available as ogr.FunctionName(). Note that ogr doesn't follow the Python naming conventions for functions.
Preparing the Work Environment

The line after the comment opens the OGR datasource (this opens the shapefile containing the data) and assigns the object to the datasource variable. Note that the path, even on Windows, uses a forward slash (/) and not a backslash.

The next line gets the first layer of the data source by its index (0). Some data sources can have many layers, but this is not the case of a Shapefile, which has only one layer. So, when working with Shapefiles, we always know that the layer of interest is layer 0.

In the final line, the print statement prints the number of features returned by layer.GetFeatureCount(). Here, we will use Python's string formatting, where the curly braces are replaced by the argument passed to format().

Now, perform the following steps:

1. In the same file, let's type the next part of our program:

```python
# Inspect the fields available in the layer.
feature_definition = layer.GetLayerDefn()
for field_index in range(feature_definition.GetFieldCount()):
    field_definition = feature_definition.GetFieldDefn(field_index)
    print(f"{field_index}", end='\t')
    print(f"{field_definition.GetTypeName()}\t{field_definition.GetName()}")
```

2. Rerun the code; you can use the Shift + F10 shortcut for this. Now, you should see the number of features as before plus a pretty table showing information on all the fields in the shapefile, as follows:

```
Number of features: 246
0 String  FIPS
1 String  ISO2
2 String  ISO3
3 Integer UN
4 String  NAME
5 Integer POP2005
6 Integer REGION
7 Integer SUBREGION
```

Process finished with exit code 0
What happens in this piece of code is that `feature_definition = layer.GetLayerDefn()` gets the object that contains the definition of the features. This object contains the definition for each field and the type of geometry.

In the `for` loop, we will get each field definition and print its index, name, and type. Note that the object returned by `layer.GetLayerDefn()` is not iterable, and we can't use `for` directly with it. So first, we will get the number of fields and use it in the `range()` function so that we can iterate through the indexes of the fields:

3. Now, type the last part, as follows:

```python
# Print a list of country names.
layer.ResetReading()
for feature in layer:
    print(feature.GetFieldAsString(4))
```

4. Run the code again and check the results in the output:

```
Number of features: 246
0 String FIPS
1 String ISO2
2 String ISO3
3 Integer UN
4 String NAME
5 Integer POP2005
6 Integer REGION
7 Integer SUBREGION
Antigua and Barbuda
Algeria
Azerbaijan
Albania
Armenia
Angola
...
Saint Barthelemy
Guernsey
Jersey
South Georgia South Sandwich Islands
Taiwan

Process finished with exit code 0
```
Preparing the Work Environment

The layer is iterable, but first, we need to ensure that we are at the beginning of the layer list with `layer.ResetReading()` (this is one of OGR's "gotcha" points).

The `feature.GetFieldAsString(4)` method returns the value of field 4 as a Python string. There are two ways of knowing whether the country names are in field 4:

- Looking at the data's DBF file (by opening it with LibreOffice or Excel)
- Looking at the table that we printed in the first part of the code

Your complete code should look similar to the following:

```python
import ogr

# Open the shapefile and get the first layer.
datasource = ogr.Open("../data/world_borders_simple.shp")
layer = datasource.GetLayerByIndex(0)
print("Number of features: ", layer.GetFeatureCount())

# Inspect the fields available in the layer.
feature_definition = layer.GetLayerDefn()
for field_index in range(feature_definition.GetFieldCount()):
    field_definition = feature_definition.GetFieldDefn(field_index)
    print("{}	{}	{}".format(field_index,
                           field_definition.GetTypeName(),
                           field_definition.GetName()))

# Print a list of country names.
layer.ResetReading()
for feature in layer:
    print(feature.GetFieldAsString(4))
```

Transforming the coordinate system and calculating the area of all countries

Now, the objective is to know how much area is occupied by each country. However, the coordinates of country borders are expressed in latitude and longitude, and we can't calculate areas in this coordinate system. We want the area to be in the metric system, so first we need to convert the spatial reference system of the geometries.
Let's also take a step further in the programming techniques and start using functions to avoid the repetition of code. Perform the following steps:

1. Create a new file in the Chapter1 directory, name this file world_areas.py, and program this first function:

   ```python
   import ogr

   def open_shapefile(file_path):
       """Open the shapefile, get the first layer and returns the ogr datasource."

       datasource = ogr.Open(file_path)
       layer = datasource.GetLayerByIndex(0)
       print("Opening {}\n".format(file_path))
       print("Number of features: {}\n".format(layer.GetFeatureCount()))
       return datasource
   
   datasource = open_shapefile("../data/world_borders_simple.shp")
   ```

2. Run the code, go to Run | Run... in the menu, and select world_areas. If everything is correct, nothing should happen. This is because we are not calling our function. Add this line of code at the end and outside the function:

   ```python
   datasource = open_shapefile("../data/world_borders_simple.shp")
   ```

3. Now, run the code again with Shift + F10 and check the output, as follows:

   ```
   Opening ../data/world_borders_simple.shp
   Number of features: 246
   ```

   Process finished with exit code 0

That's wonderful! You just created a piece of very useful and reusable code. You now have a function that can open any shapefile, print the number of features, and return the ogr datasource. From now on, you can reuse this function in any of your projects.

You are already familiar with how this code works, but there are a few novelties here that deserve an explanation. The `def` statement defines a function with the `def function_name(arguments):` syntax.

Remember when I told you that OGR doesn't follow Python's naming convention? Well, the convention is that function names should all be in lowercase with an underscore between words. A good hint for names is to follow the **verb_noun** rule.
Preparing the Work Environment

These conventions are described in a document called PEP-8, where PEP stands for Python Enhancement Program. You can find this document at https://www.python.org/dev/peps/pep-0008/.

Right after the function's definition, you can see a description between triple quotes; this is a docstring, and it is used to document the code. It's optional but very useful for you to know what the function does.

Now, let's get back to our code. The second important thing to point out is the return statement. This makes the function return the values of the variables listed after the statement—in this case, the datasource.

It's very important that all pieces of the OGR objects flow together through the program. In this case, if we return only the layer, for example, we will get a runtime error later in our program. This happens because in OGR internals, the layer has a reference to the data source, and when you exit a Python function, all objects that don't exit the function are trashed, and this breaks the reference.

Now, the next step is to create a function that performs the transformation. In OGR, the transformation is made in the feature's geometry, so we need to iterate over the features, get the geometry, and transform its coordinates. We will do this using the following steps:

1. Add the following function to your world_areas.py file just after the open_shapefile function:
   ```python
   def transform_geometries(datasource, src_epsg, dst_epsg):
       """Transform the coordinates of all geometries in the first layer."
       """
       # Part 1
       src_srs = osr.SpatialReference()
       src_srs.ImportFromEPSG(src_epsg)
       dst_srs = osr.SpatialReference()
       dst_srs.ImportFromEPSG(dst_epsg)
       transformation = osr.CoordinateTransformation(src_srs, dst_srs)
       layer = datasource.GetLayerByIndex(0)

       # Part 2
       geoms = []
   ```
layer.ResetReading()
for feature in layer:
    geom = feature.GetGeometryRef().Clone()
    geom.Transform(transformation)
    geoms.append(geom)
return geoms

The function takes three arguments: the ogr layer, the EPSG code of the coordinate system of the file, and the EPSG code for the transformation output. Here, it created an osr.CoordinateTransformation object; this object contains the instructions to perform the transformation. Probably by now, Pycharm should be complaining that osr is an unresolved reference; osr is the part of GDAL that deals with coordinate systems.

2. Now, import the module by adding this line at the top of your code:
   ```python
   import osr
   ```
   Here, the code iterates over all features, gets a reference to the geometry, and performs the transformation. As we don't want to change the original data, the geometry is cloned, and the transformation is made on the clone.
   Python lists are ordered; this means that the elements are in the same order in which they are appended to the list, and this order is always kept. This allows us to create a list of geometries in the same order of the features that are in the data source. This means that the geometries in the list and the features have the same index and can be related in the future by the index.

3. Now, let's test the code; add the following lines at the end of the file (the first line is the one that you already added before):
   ```python
   datasource = open_shapefile("../data/world_borders_simple.shp")
   layer = datasource.GetLayerByIndex(0)
   feature = layer.GetFeature(0)
   print("Before transformation:")
   print(feature.GetGeometryRef())
   transformed_geoms = transform_geometries(datasource, 4326, 3395)
   print("After transformation:")
   print(transformed_geoms[0])
   ```

4. Finally, before you run the code, add one more import at the beginning of the program. It should be the first statement of your code, as follows:
   ```python
   from __future__ import print_function
   ```
   This import allows us to use the print() function from Python 3 with the desired behavior, thus maintaining the compatibility.
Preparing the Work Environment

5. The complete code should look similar to this:

```python
from __future__ import print_function
import ogr
import osr

def open_shapefile(file_path):
    ...

def transform_geometries(datasource, src_epsg, dst_epsg):
    ...

datasource = open_shapefile("../data/world_borders_simple.shp")
layer = datasource.GetLayerByIndex(0)
feature = layer.GetFeature(0)
print("Before transformation:")
print(feature.GetGeometryRef())
transformed_geoms = transform_geometries(datasource, 4326, 3395)
print("After transformation:")
print(transformed_geoms[0])
```

6. Run your program again by pressing Shift + F10. In the output, note the difference in the coordinates before and after the transformation:

```
Opening ../data/world_borders_simple.shp
Number of features: 246
Before transformation:
MULTI POLYGON (((-61.686668 17.024441000000138 ... )))
After transformation:
MULTI POLYGON (((-6866928.4704937246 ... )))
```

7. Now, add another function. This function will calculate the area in square meters (because we will use the geometries that have coordinates in meters), convert the value (or not) to square kilometers or square miles, and store the values in another list with the same order as before. Execute the following code:

```python
def calculate_areas(geometries, unity='km2'):
    """Calculate the area for a list of ogr geometries.""
    conversion_factor = {
        'sqmi': 2589988.11,
        'km2': 1000000,
```
Chapter 1

{'m': 1}

# Part 2
if unity not in conversion_factor:
    raise ValueError(
        "This unity is not defined: {}").format(unity))

# Part 3
areas = []
for geom in geometries:
    area = geom.Area()
    areas.append(area / conversion_factor[unity])
return areas

Firstly, note that in the function definition, we use `unity='km2';` this is a keyword argument, and when you call the functions, this argument is optional.

In Part 1, a dictionary is used to define a few conversion factors for the area unit. Feel free to add more units if you wish. By the way, Python doesn't care if you use single or double quotes.

In Part 2, a verification is made to check whether the passed unity exists and whether it is defined in `conversion_factor`. Another way of doing this is catching the exception later; however, for now, let's opt for readability.

In Part 3, the code iterates the `ogr` geometries, calculates the area, converts the values, and puts it on a list.

8. Now, to test the code, edit your first line, including division to the future imports. This will ensure that all divisions return floating point numbers and not integers. Here's how it should look:

```python
from __future__ import print_function, division
```

9. Then, update the testing part of your code to the following:

```python
datasource = open_shapefile("../data/world_borders_simple.shp")
transformed_geoms = transform_geometries(datasource, 4326, 3395)
calculated_areas = calculate_areas(transformed_geoms, unity='sqmi')
print(calculated_areas)
```

10. Run it, change the unity, then run again, and note how the results change.

Very well, unity conversion is another very important procedure in geoprocessing, and you just implemented it in your `calculate_areas` function.

However, having a list of numbers as the output is not very useful to us. So, it's time to combine everything that we did so far in order to extract valuable information.
Preparing the Work Environment

Sort the countries by area size

You programmed three functions so far; now, let's add another one to our list by converting the code that generated a list of country names to a function and add this function to world_areas.py, as follows:

```python
def get_country_names(datasource):
    """Returns a list of country names."""
    layer = datasource.GetLayerByIndex(0)
    country_names = []
    layer.ResetReading()
    for feature in layer:
        country_names.append(feature.GetFieldAsString(4))
    return country_names
```

Now, we have four functions, which are:

- open_shapefile
- transform_geometries
- calculate_areas
- get_country_names

All these functions return iterables, with each item sharing the same index on all of them, thus making it easy to combine the information.

So, let's take advantage of this feature to sort the countries by area size and return a list of the five biggest countries and their areas. For this, add another function, as follows:

```python
def get_biggest_countries(countries, areas, elements=5):
    """Returns a list of n countries sorted by area size."""
    countries_list = [list(country)
                      for country in zip(areas, countries)]
    sorted_countries = sorted(countries_list,
                               key=itemgetter(0), reverse=True)
    return sorted_countries[:5]
```

In the first line, the two lists are zipped together, producing a list of country-area pairs. Then, we used the Python list's sorted method, but as we don't want the lists to be sorted by both values, we will define the key for sorting. Finally, the list is sliced, returning only the desired number of values.

1. In order to run this code, you need to import the `itemgetter` function and put it at the beginning of the code but after `from __future__ imports`, as follows:
   ```python
   from operator import itemgetter
   ```
2. Now, edit the testing part of your code to look similar to the following:

```python
datasource = open_shapefile("../data/world_borders_simple.shp")
transformed_geoms = transform_geometries(datasource, 4326, 3395)
country_names = get_country_names(datasource)
country_areas = calculate_areas(transformed_geoms)
biggest_countries = get_biggest_countries(country_names, country_areas)

for item in biggest_countries:
    print("{}\t{}").format(item[0], item[1])
```

3. Now, run the code and take a look at the results, as follows:

```
Opening ../data/world_borders_simple.shp
Number of features: 246
82820725.1423 Russia
51163710.3726 Canada
35224817.514 Greenland
21674429.8403 United States
14851905.8596 China
```

Process finished with exit code 0

**Summary**

In this chapter, we had a brief introduction to the libraries and packages that we will use in this book. By installing these libraries, you also learned the general procedure of how to search and install Python packages. You can use this procedure in other cases whenever you feel the need for other libraries in your applications.

Then, we wrote code that made use of the OGR library to open a shapefile and perform area calculation and sorting. These simple procedures showed a little bit of the internal organization of OGR, how it handles geographic data, and how it is possible to extract information from them. In the next chapter, we will use some of the techniques learned here to read data and process vector points.
Where to buy this book

You can buy Geospatial Development By Example with Python 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.