3 Reading and writing vector data

This chapter covers

- Understanding vector data
- Introducing OGR
- Reading vector data
- Creating new vector datasets
- Updating existing datasets

They seem to be rare these days, but you've probably seen a paper roadmap designed to be folded up and kept in your car. Unlike the more recent web maps that we're used to using, these maps don't use aerial imagery. Instead, features on the maps are all drawn as geometric objects—namely, points, lines, and polygons. These types of data, where the geographic features are all distinct objects, are called *vector datasets*.

Unless you only plan to look at maps that someone else has made, you'll need to know how to read and write these types of data. If you want to work with existing data in any way, whether you're summarizing, editing, deriving new data, or performing sophisticated spatial analyses, you need to read it in from a file first. You also need to write any new or modified data back out to a disk. For example, if you

had a nationwide city dataset but needed to analyze only data from cities with 100,000 people or more, you could extract those cities out of your original dataset and run your analysis on them while ignoring the smaller towns. Optionally, you could also save the smaller dataset to a new file for later use.

In this chapter you'll learn basic ideas behind vector data and how to use the OGR library to read, write, and edit these types of datasets.

3.1 Introduction to vector data

At its most basic, *vector data* are data in which geographic features are represented as discrete geometries—specifically, points, lines, and polygons. Geographic features that have distinct boundaries, such as cities, work well as vector data, but continuous data, such as elevation, don't. It would be difficult to draw a single polygon around all areas with the same elevation, at least if you were in a mountainous area. You could, however, use polygons to differentiate between different elevation ranges. For example, polygons showing subalpine zones for a region would be a good proxy for an elevation range, but you'd lose much of the detailed elevation data within those polygons. Many types of data are excellent candidates for a vector representation, though, such as features in the roadmap mentioned earlier. Roads are represented as lines, counties and states are polygons, and depending on the scale of the map, cities are drawn as either points or polygons. In fact, all of the features on the map are probably represented as points, lines, or polygons.

The type of geometry used to draw a feature can be dependent on scale, however. Figure 3.1 shows an example of this. On the map of New York State, cities are shown as points, major roads as lines, and counties as polygons. A map of a smaller area, such as New York City, will symbolize features differently. In this case, roads are still lines, but the city and its boroughs are polygons instead of points. Now points would be used to represent features such as libraries or police stations.

You can imagine many other examples of geographic data that lend themselves to being represented this way. Anything that can be described with a single set of coordinates, such as latitude and longitude, can be represented as a point. This includes cities, restaurants, mountain peaks, weather stations, and geocache locations. In addition to their x and y coordinates (such as latitude and longitude), points can have a third z coordinate that represents elevation.

Geographic areas with closed boundaries can be represented as polygons. Examples are states, lakes, congressional districts, zip codes, and land ownership, along with many of the same features that can be symbolized as points such as cities and parks. Other features that could be represented as polygons, but probably not as points, include countries, continents, and oceans.

Linear features, such roads, rivers, power lines, and bus routes, all lend themselves to being characterized as lines. Once again, however, scale can make a difference. For example, a map of New Orleans could show the Mississippi River as a polygon rather than a line because it's so wide. This would also allow the map to show the irregular banks of the river, rather than just a smooth line, as shown in figure 3.2.

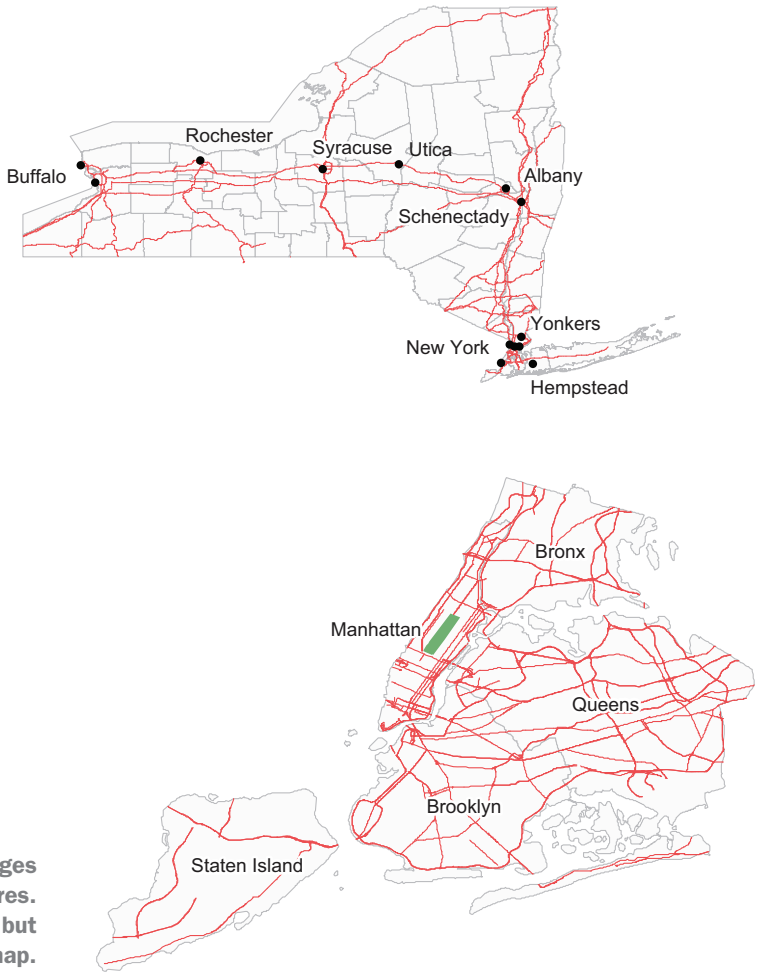


Figure 3.1 An example of how scale changes the geometries used to draw certain features. New York City is a point on the state map, but is made of several polygons on the city map.

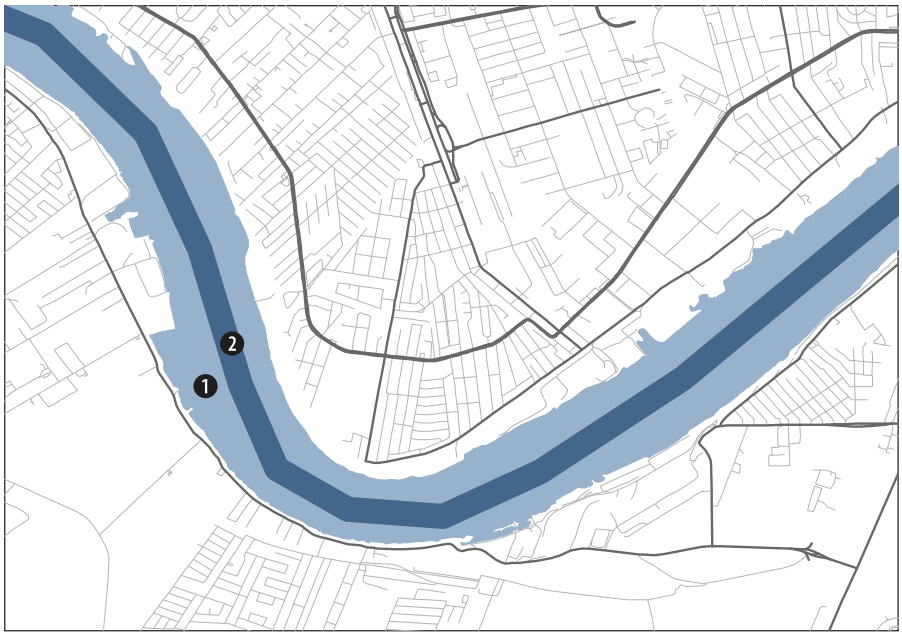
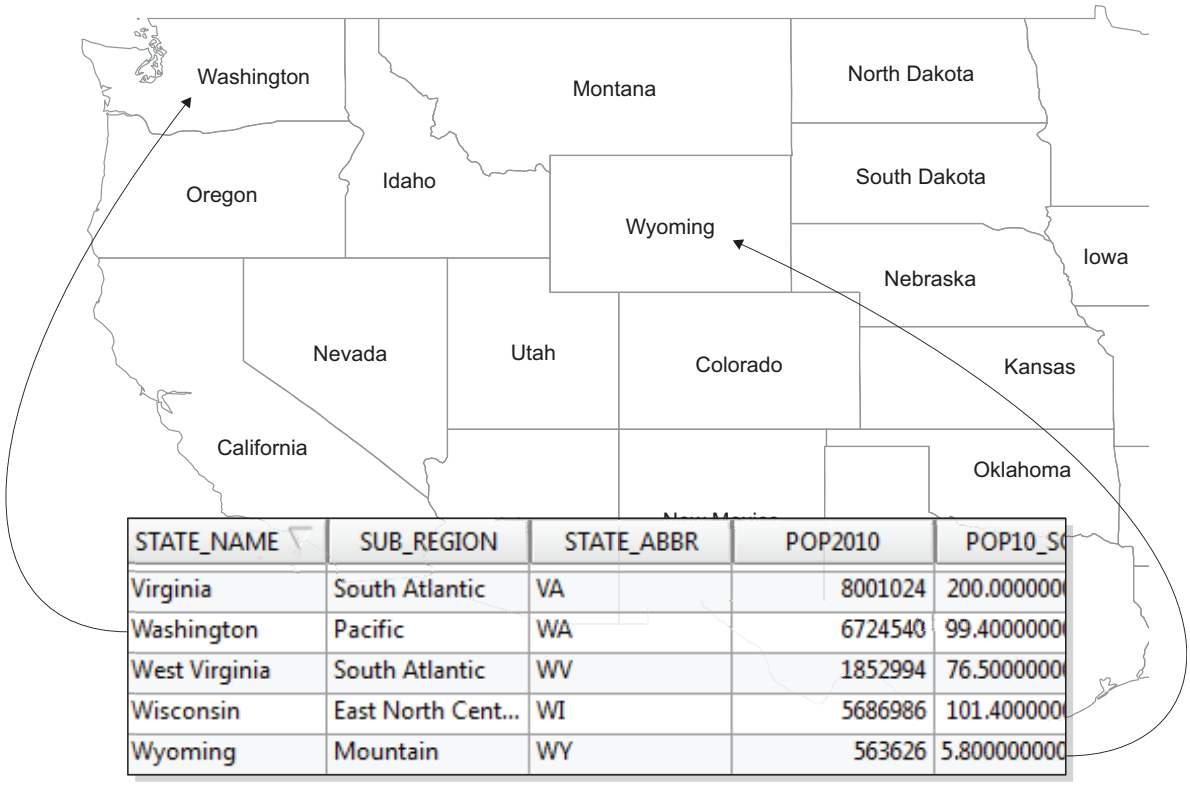


Figure 3.2 The difference between using polygon 1 and line 2 geometries to represent the Mississippi River. The polygon shows the details along the banks, while the line doesn't.



STATE_NAME	SUB_REGION	STATE_ABBR	POP2010	POP10_S
Virginia	South Atlantic	VA	8001024	200.000000
Washington	Pacific	WA	6724540	99.4000000
West Virginia	South Atlantic	WV	1852994	76.5000000
Wisconsin	East North Cent...	WI	5686986	101.400000
Wyoming	Mountain	WY	563626	5.800000000

Figure 3.3 An attribute table for a dataset containing state boundaries within the United States. Each state polygon has an associated row in the data table with several attributes, including state name and population in 2010.

Vector data is more than geometries, however. Each one of these features also has associated attributes. These attributes can relate directly to the geometry itself, such as the area or perimeter of a polygon, or length of a line, but other attributes may be present as well. Figure 3.3 shows a simple example of a states dataset that stores the state name, abbreviation, population, and other data along with each feature. As you can see from the figure, these attributes can be of various types. They can be numeric, such as the city population or road speed limit, strings like city or road names, or dates such as the date the land parcel was purchased or last appraised. Certain types of vector data also support BLOBs (binary large objects), which can be used to store binary data such as photographs.

It should be clear by now that this type of data is well suited for making maps, but some reasons might not be so obvious. One example is how well it scales when drawing. If you're familiar with web graphics, you probably know that vector graphics such as SVG (scalable vector graphics) work much better than raster graphics such as PNG when displayed at different scales. Even if you know nothing about SVG, you've surely seen an image on a website that's pixelated and ugly. That's a raster graphic displayed at a higher resolution than it was designed for. This doesn't happen with vector graphics,

and the exact same principle applies to vector GIS data. It always looks smooth, no matter the scale.

That doesn't mean that scale is irrelevant, though. As you saw earlier, scale affects the type of geometry used to represent a geographic feature, but it also affects the resolution you should use for a feature. A simple way to think of resolution is to equate it to detail. The higher the resolution, the more detail can be shown. For example, a map of the United States wouldn't show all of the individual San Juan Islands off the coast of Washington State, and in fact, the dataset wouldn't even need to include them. A map of only Washington State, however, would definitely need a higher-resolution dataset that includes the islands, as seen in figure 3.4. Keep in mind that resolution isn't important only for display, but also for analysis. For example, the two maps of Washington would provide extremely different measurements for coastline length.

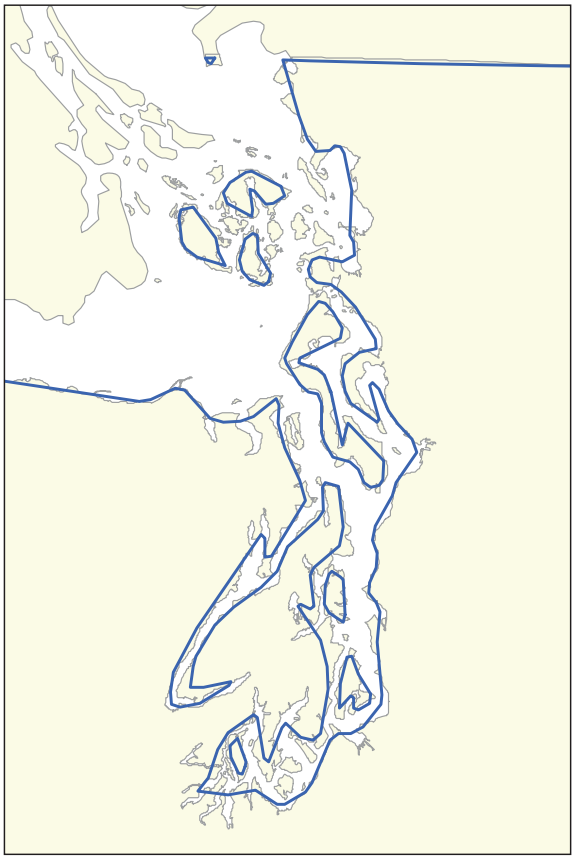


Figure 3.4 An example showing the difference that resolution makes. The dataset shown with the thick outline has a lower resolution than the one shown with shading. Notice the difference in the amount of detail available in the two datasets.

The coastline paradox

Have you ever thought about how to measure the coastline of a landmass? As first pointed out by the English mathematician Lewis Fry Richardson, this isn't as easy as you might think, because the final measurement depends totally on scale. For example, think about a wild section of coastline with multiple headlands, with a road running along beside it. Imagine that you drive along that road and use your car's odometer to measure the distance, and then you get out of the car and walk back the way you came. But when on foot, you walk out along the edges of the headlands and follow other curves in the coast that the road doesn't. It should be easy to imagine that you'd walk farther than you drove because you took more detours. The same principle applies when measuring the entire coastline, because you can measure more variation if you measure in smaller increments. In fact, measuring the coast of Great Britain in 50-km increments instead of 100-km increments increases the final measurement by about 600 km. You can see another example of this, using part of Washington State,

(continued)

in figure 3.3. If you were to measure all of the twists and turns in the higher-resolution dataset, you'd get a longer coastline measurement than if you measured the lower-resolution coastline shown by the dark line, which doesn't even include many of the islands.

As mentioned previously, vector data isn't only for making maps. In fact, I couldn't make a pretty map if my life depended on it, but I do know a little bit more about data analysis. One common type of vector data analysis is to measure relationships between geographic features, typically by overlaying them on one another to determine their spatial relationship. For example, you could determine if two features overlap spatially and what that area of overlap is. Figure 3.5 shows the New Orleans city boundaries overlaid on a wetlands dataset. You could use this information to determine where wetlands exist within the city of New Orleans and how much of the city's area is or isn't wetland.

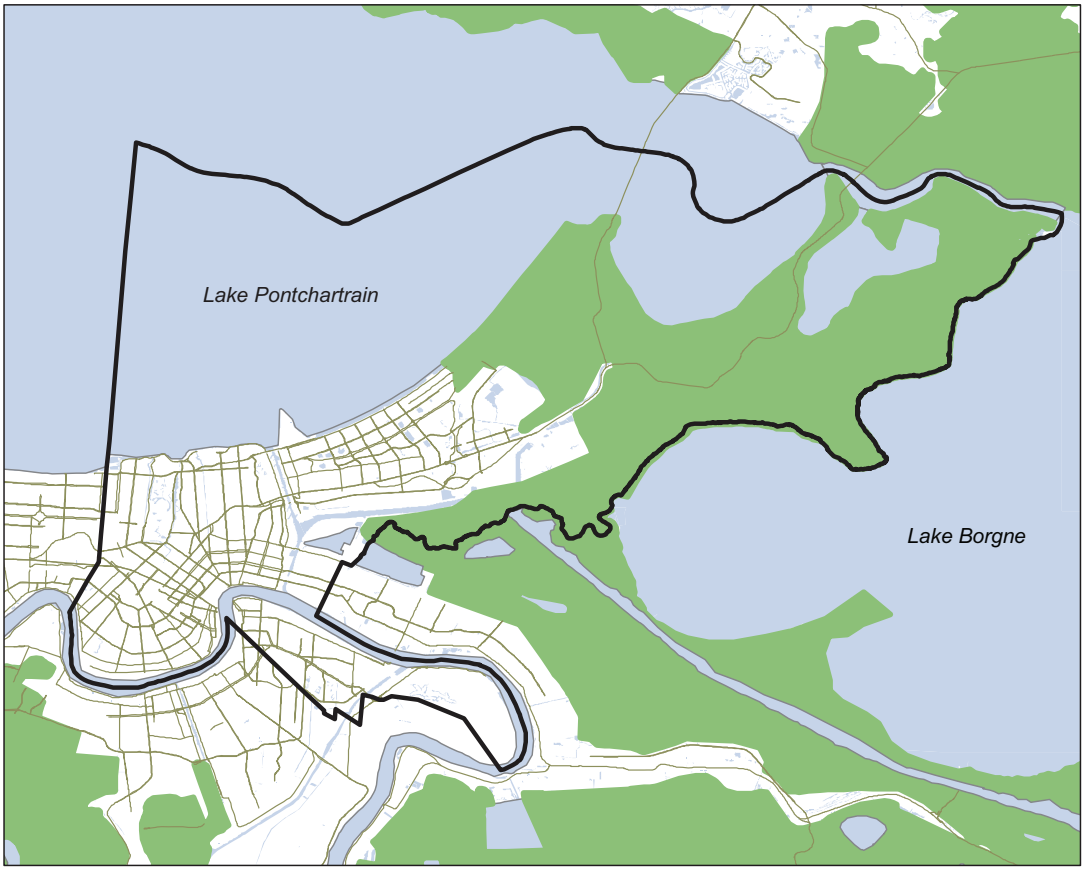


Figure 3.5 An example of a vector overlay operation. The dark outline is the City of New Orleans boundary, and the darker land areas are wetlands. These two datasets could be used to determine the percentage of land area within the New Orleans boundary that is wetlands.

Another aspect of spatial relationships is the distance between two features. You could find the distance between two weather stations, or all of the sandwich shops within one mile of your office. I helped out with a study a few years ago in which the researchers needed both distances and spatial relationships. They needed to know how far GPS-collared deer traveled between readings, but also the direction of travel and how they interacted with man-made features such as roads. One question in particular was if they crossed the roads, and if so, how often.

Speaking of roads, vector datasets also do a good job of representing networks, such as road networks. A properly configured road network can be used to find routes and drive times between two locations, similar to the results you see on various web-mapping sites. Businesses can also use information like this to provide services. For example, a pizza joint might use network analysis to determine which parts of town they can reach within a 15-minute drive to set their delivery area.

As with other types of data, you have multiple ways to store vector data. Similar to the way you can store a photograph as a JPEG, PNG, TIFF, bitmap, or one of many other file types, many different file formats can be used for storing vector data. I'll talk more about the possibilities in the next chapter, but for now I'll briefly mention a few common formats, several of which we'll use in this chapter.

Shapefiles are a popular format for storing vector data. A shapefile isn't made of a single file, however. In fact, this format requires a minimum of three binary files, each of which serves a different purpose. Geometry information is stored in .shp and .shx files, and attribute values are stored in a .dbf file. Additionally, other data, such as indexes or spatial reference information, can be stored in even more files. Generally you don't need to know anything about these files, but you do need to make sure that they're all kept together in the same folder.

Another widely used format, especially for web-mapping applications, is GeoJSON. These are plain text files that you can open up and look at in any text editor. Unlike a shapefile, a GeoJSON dataset consists of one file that stores all required information.

Vector data can also be stored in relational databases, which allows for multiuser access as well as various types of indexing. Two of the most common options for this are spatial extensions built for widely used database systems. The PostGIS extension runs on top of PostgreSQL, and SpatiaLite works with SQLite databases. Another popular database format is the Esri file geodatabase, which is completely different in that it isn't part of an existing database system.

3.2 Introduction to OGR

The OGR Simple Features Library is part of the Geospatial Data Abstraction Library (GDAL), an extremely popular open source library for reading and writing spatial data. The OGR portion of GDAL is the part that provides the ability to read and write many different vector data formats. OGR also allows you to create and manipulate geometries; edit attribute values; filter vector data based on attribute values or spatial location; and it also offers data analysis capabilities. In short, if you want to use GDAL to work with vector data, OGR is what you need to learn about, and you will, in the next four chapters.

The GDAL library was originally written in C and C++, but it has bindings for several other languages, including Python, so there's an interface to the GDAL/OGR library from Python, not that the code was rewritten in Python. Therefore, to use GDAL with Python, you need to install both the GDAL library and the Python bindings for it. If you haven't yet done this, please see appendix A for detailed installation instructions.

NOTE What does the OGR acronym stand for, anyway? It used to stand for OpenGIS Simple Features Reference Implementation, but because OGR isn't fully compliant with the OpenGIS Simple Features specification, the name was changed and now the OGR part of it doesn't stand for anything and is only historical in nature.

Several functions used in this chapter are from the ospybook Python module available for download at www.manning.com/books/geoprocessing-with-python. You'll want to install this module, too. The sample datasets are available from the same site.

Before you start working with OGR, it's useful to look at how various objects in the OGR universe are related to each other, as shown in figure 3.6. If you don't understand this hierarchy, then the steps required to read and write data won't make much sense. When you use OGR to open a data source, such as a shapefile, GeoJSON file, SpatiaLite, or PostGIS database, you'll have a `DataSource` object. This data source can have one or more child `Layer` objects, one for each dataset contained in the data source. Many vector data formats, such as the shapefile examples used in this chapter, can only contain one dataset. But others, such as SpatiaLite, can contain multiple datasets, and you'll see examples of this in the next chapter. Regardless of how many datasets are in a data source, each one is considered a layer by OGR. Even several of my students, who use GIS regularly for their classes and research, get confused by this if they mostly use shapefiles, because it's counterintuitive to them that something called a layer sits between the data source and the actual data.

And speaking of the actual data, each layer contains a collection of `Feature` objects that holds the geometries and their attributes. If you load vector data into a GIS, such as QGIS, and then look at the attribute table, you'll see something similar to

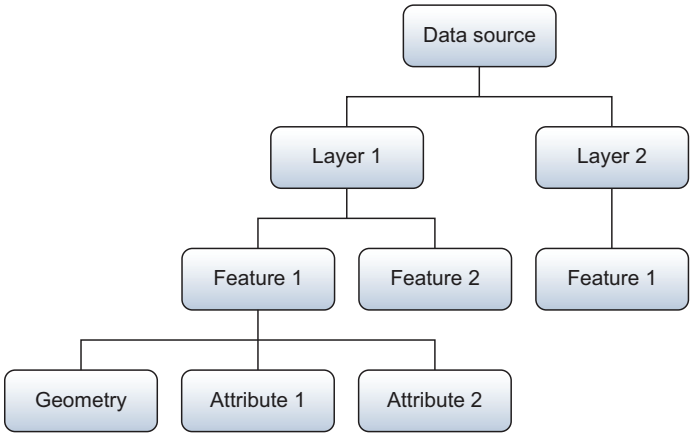


Figure 3.6 The OGR class structure. Each data source can have multiple layers, each layer can have multiple features, and each feature contains a geometry and one or more attributes.

	scalerank	featurecla	LABELRANK	SOVEREIGNT	SOV_A3	ADM0_DIF	LEVEL	TYPE	
0	3	Admin-0 country	5.00	Netherlands	NL1	1.00	2.00	Country	Arub
1	0	Admin-0 country	3.00	Afghanistan	AFG	0.00	2.00	Sovereign coun...	Afgh
2	0	Admin-0 country	3.00	Angola	AGO	0.00	2.00	Sovereign coun...	Angc
3	3	Admin-0 country	6.00	United Kingd...	GB1	1.00	2.00	Dependency	Angu
4	0	Admin-0 country	6.00	Albania	ALB	0.00	2.00	Sovereign coun...	Alba

Figure 3.7 An example of an attribute table shown in QGIS. Each row corresponds to a feature, and each column is an attribute field.

figure 3.7. Each row in the table corresponds to a feature, such as the feature representing Afghanistan. Each column corresponds to an attribute field, and in this case two of the attributes are `SOVEREIGNT` and `TYPE`. Although you can open data tables that don't have any spatial information or geometries associated with the features, we'll work with datasets that do have geometries. As you can see in figure 3.7, the geometries don't show up in the attribute table in QGIS, although other GIS software packages, such as ArcGIS, do show a shape column in the attribute table.

The first step to accessing any vector data is to open the data source. For this, you need to have an appropriate driver that tells OGR how to work with your data format. The GDAL/OGR website lists more than 70 vector formats that OGR is capable of reading, although it can't write to all of them. Each one of these has its own driver. It's likely that your version of OGR doesn't support all of those listed, but you can always compile it yourself if you need something that's missing (note that this is easier said than done in many cases). See www.gdal.org/ogr_formats.html for the list of all available formats and specific details pertaining to each one.

DEFINITION A driver is a translator for a specific data format, such as GeoJSON or shapefile. It tells OGR how to read and write that particular format. If no driver for a format is compiled into OGR, then OGR can't work with it.

If you aren't sure if your installation of GDAL/OGR supports a particular data format, you can use the ogrinfo command-line utility to find out which drivers are available. The location of this utility on your computer depends on your operating system and how you installed GDAL, so you might need to refer back to appendix A. If you aren't used to using a command line, you may be tempted to double-click the ogrinfo executable file, but that won't get you anywhere useful. Instead, you need to run `ogrinfo` from a terminal window or Windows command prompt. At any rate, once you find the executable, you'll want to run it with the `--formats` option. Figure 3.8 shows an example of running it on my Windows 7 machine, although I've cut off most of the output.

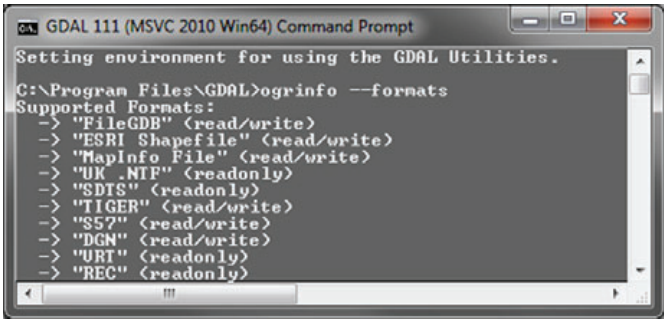


Figure 3.8 An example of running the ogrinfo utility from a GDAL command prompt on a Windows computer

Figure 3.9 Sample Python interactive session showing how to get drivers

As you can see, ogrinfo not only tells you which drivers are included with your version of OGR, but also whether it can write to each one as well as read from it.

> **TIP** Information about vector formats supported by OGR can be found at www.gdal.org/ogr_formats.html.

You can also determine which drivers are available using Python. In fact, let's try it. Start by opening up your favorite Python interactive environment. I'll use IDLE (figure 3.9) because it's the one that's packaged with Python, but you can use whichever one you're comfortable with. The first thing you need to do is import the `ogr` module so that you can use it. This module lives inside the `osgeo` package, which was installed when you installed the Python bindings for GDAL. All of the modules in this package are named with lowercase letters, which is how you need to refer to them in Python. Once you've imported `ogr`, then you can use `ogr.GetDriverByName` to find a specific driver:

```
from osgeo import ogr
driver = ogr.GetDriverByName('GeoJSON')
```

Use the name from the Code column on the OGR Vector Formats webpage. If you get a valid driver and print it out, you'll see information about where the object is stored in memory. The important thing is that there was something for it to print out because it means you successfully found a driver. If you pass an invalid name, or the name of a missing driver, the function will return `None`. See figure 3.9 for examples.

A function called `print_drivers` in the ospybook module will also print out a list of available drivers. This is shown in figure 3.9.

3.3 Reading vector data

Now that you know what formats are available to work with, it's time to read data. You'll start with a cities shapefile, the ne_50m_populated_places.shp dataset in the global

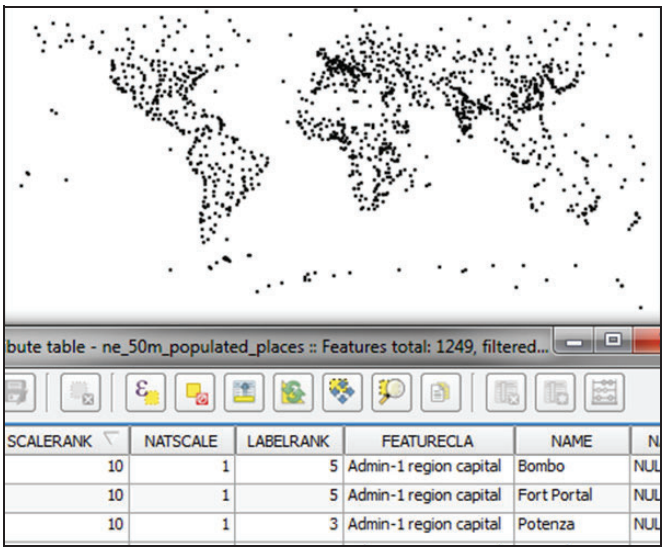


Figure 3.10 The geometries and attributes from ne_50m_populated_places.shp as seen in QGIS

subfolder of your osgeopy-data folder. Feel free to open it up in QGIS and look. Not only will you see the cities shown in figure 3.10, but you'll also see that the attribute table contains a collection of fields, most of which aren't visible in the screenshot.

Listing 3.1 shows a little script that prints out the names, populations, and coordinates for the first 10 features in this dataset. Don't worry if it doesn't make much sense at first glance because we'll go over it in excruciating detail in a moment. The file is included with the source code for this chapter, so if you want to try it out, you can open it in IDLE, change the filename in the third line of code to match your setup, and then choose Run Module under the Run menu.

Listing 3.1 Printing data from the first ten features in a shapefile

```
import sys
from osgeo import ogr

fn = r'D:\osgeopy-data\global\ne_50m_populated_places.shp'
ds = ogr.Open(fn, 0)
if ds is None:
    sys.exit('Could not open {0}.'.format(fn))
lyr = ds.GetLayer(0)

i = 0
for feat in lyr:
    pt = feat.geometry()
    x = pt.GetX()
    y = pt.GetY()
    name = feat.GetField('NAME')
    pop = feat.GetField('POP_MAX')
    print(name, pop, x, y)
    i += 1
    if i == 10:
        break
del ds
```

Don't forget to import ogr

Open the data source

Get x, y coordinates

Get attribute values

The basic outline is simple. The first thing you do is open the shapefile and make sure that the result of that operation isn't equal to `None`, because that would mean the data source couldn't be opened. I tend to call this variable `ds`, short for data source. After making sure the file is opened, you retrieve the first layer from the data source. Then you iterate through the first 10 features in the layer and for each one, get the geometry object, its coordinates, and the `NAME` and `POP_MAX` attribute values. Then you print the information about the feature before moving on to the next one. When done, you delete the `ds` variable to force the file to close.

If you successfully ran the code, you should have 10 lines of output that look something like this, although you won't have the parentheses if using Python 3:

```
('Bombo', 75000, 32.533299524864844, 0.5832991056146284)
('Fort Portal', 42670, 30.27500161597942, 0.671004121125236)
<snip>
('Clermont-Ferrand', 233050, 3.080008095928406, 45.779982115759424)
```

Let's look at this in a little more detail. You open a data source by passing the filename and an optional update flag to the `Open` function. This is a standalone function in the OGR module, so you prefix the function name with the module name so that Python can find it. If the second parameter isn't provided it defaults to 0, which will open the file in read-only mode. You could have passed `1` or `True` to open it in update, or edit, mode instead.

If the file can't be opened, then the `Open` function returns `None`, so the next thing you do is check for this and print out an error message and quit if needed. I like to check for this so I can solve the problem immediately and in the manner of my choosing (quitting, in this case) instead of waiting for the script to crash when it tries to use the nonexistent data source. Change the filename in listing 3.1 to a bogus one and run the script if you want to see this behavior in action:

```
fn = r'D:\osgeopy-data\global\ne_50m_populated_places.shp'
ds = ogr.Open(fn, 0)
if ds is None:
    sys.exit('Could not open {0}.'.format(fn))
lyr = ds.GetLayer(0)
```

Remember that data sources are made of one or more layers that hold the data, so after opening the data source you need to get the layer from it. Data sources have a function called `GetLayer` that takes either a layer index or a layer name and returns the corresponding `Layer` object inside that particular data source. Layer indexes start at 0, so the first layer has index 0, the second has index 1, and so on. If you don't provide any parameters to `GetLayer`, then it returns the first layer in the data source. The shapefile only has one layer, so the index isn't technically needed in this case.

Now you want to get the data out of your layer. Recall that each layer is made of one or more features, with each feature representing a geographic object. The geometries and attribute values are attached to these features, so you need to look at them to get your data. The second half of the code in listing 3.1 loops through the first 10

features in the layer and prints information about each one. Here's the interesting part of it again:

```
for feat in lyr:
    pt = feat.geometry()
    x = pt.GetX()
    y = pt.GetY()
    name = feat.GetField('NAME')
    pop = feat.GetField('POP_MAX')
    print(name, pop, x, y)
```

The layer is a collection of features that you can iterate over with a `for` loop. Each time through the loop, the `feat` variable will be the next feature in the layer, and the loop will iterate over all features in the layer before stopping. You don't want to print out all 1,249 features, though, so you force it to stop after the first 10.

The first thing you do inside the loop is get the geometry from the feature and stick it in a variable called `pt`. Once you have the geometry, you grab its x and y coordinates and store them in variables to use later.

Next you retrieve the values from the `NAME` and `POP_MAX` fields and store those in variables as well. The `GetField` function takes either an attribute name or index and returns the value of that field. Once you have the attributes, you print out all of the information you gathered about the current feature.

One thing you should be aware of is that the `GetField` function returns data that's the same data type as that in the underlying dataset. In this example, the value in the `name` variable is a string, but the value stored in `pop` is a number. If you want the data in another format, check out appendix B to see a list of functions that return values as a specific type. For example, if you wanted `pop` to be a string so that you could concatenate it to another string, you could use `GetFieldAsString`.

```
pop = feat.GetFieldAsString('POP_MAX')
```

Note that not all data formats support all field types, and not all data can successfully be converted between types, so you should test things thoroughly before relying on these automatic conversions. Not only are these functions useful for converting data between types, but you can also use them to make data types more evident in your code. For example, if you use `GetFieldAsInteger`, then it's obvious to anyone reading your code that the value is an integer.

3.3.1 Accessing specific features

Sometimes you don't need every feature, so you have no reason to iterate through all of them as you've done so far. One powerful method of limiting features to a subset is to select them by attribute value or spatial extent, and you'll do that in chapter 5. Another way is to look at features with specific offsets, also called *feature IDs* (FIDs). The offset is the position that the feature is at in the dataset, starting with zero. It depends entirely on the position of the feature in the file and has nothing to do with the sort order in memory. For example, if you open the ne_50m_populated _places shapefile in

A. Native sort order

	NAME
0	Bombo
1	Fort Portal
2	Potenza
3	Campobasso
4	Aosta
5	Mariehamn
6	Ramallah
7	Vatican City

B. Sorted by name

	NAME
346	Abakan
908	Abeche
1163	Abidjan
83	Aboa Station
772	Abu Dhabi
878	Abuja
859	Acapulco
1147	Accra

Figure 3.11 The attribute table for the ne_50m_populated_places shapefile. Table A shows the native sort order, with the FIDs in order. Table B has been sorted by city name, and the FIDs are no longer ordered sequentially.

QGIS and look at the attribute table, it would show Bombo as the first record in the table, as in figure 3.11A. See the numbers in the left-most column? Those are the offset values. Now try sorting the table by name by clicking on the NAME column header, as shown in figure 3.11B. Now the first record shown in the table is the one for Abakan, but it has an offset of 346. As you can see, that left-most column isn't a row number like you see in spreadsheets, where the row numbers are always in the right order no matter how you sort the data. These numbers represent the order in the file instead.

If you know the offset of the feature you want, you can ask for that feature by FID. To get the feature for Vatican City, you use `GetFeature(7)`.

You can also get the total number of features with `GetFeatureCount`, so you could grab the last feature in the layer like this:

```
>>> num_features = lyr.GetFeatureCount()
>>> last_feature = lyr.GetFeature(num_features - 1)
>>> last_feature.NAME
'Hong Kong'
```

You have to subtract one from the total number of features because the first index is zero. If you had tried to get the feature at index `num_features`, you'd have gotten an error message saying that the feature ID was out of the available range. This snippet also shows an alternate way of retrieving an attribute value from a feature, instead of using `GetField`, but it only works if you know the names beforehand so that you can hardcode them into your script.

THE CURRENT FEATURE

Another important point is that the functions that return features keep track of which feature was last accessed; this is the *current feature*. When you first get the layer object, it has no current feature. But if you start iterating through features, the first time through the loop, the current feature is the one with an FID of zero. The second time through the loop, the current feature is the one with offset 1, and so on. If you use `GetFeature` to get the one with an FID of 5, that's now the current feature, and if you then call `GetNextFeature` or start a loop, the next feature returned will be the one with offset 6. Yes, you read that right. If you iterate through the features in the layer, it doesn't start at the first one if you've already set the current feature.

Based on what you've learned so far, what do you think would happen if you iterated through all of the features and printed out their names and populations, but then later tried to iterate through a second time to print out their names and coordinates? If you

guessed that no coordinates would print out, you were right. The first loop stops when it runs out of features, so the current feature is pointing past the last one and isn't reset to the beginning (see figure 3.12). No next feature is there when the second loop starts, so nothing happens. How do you get the current feature to point to the beginning again? You wouldn't want to use a FID of zero, because if you tried to iterate through them all, the first feature would be skipped. To solve this problem, use the `layer.ResetReading()` function, which sets the current feature pointer to a location before the first feature, similar to when you first opened the layer.

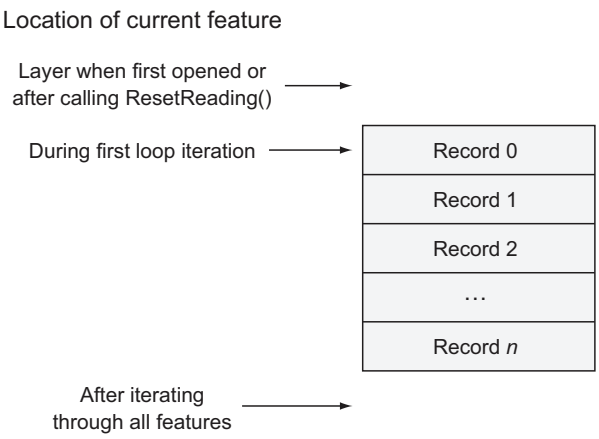


Figure 3.12 The location of the current feature pointer at various times

3.3.2 Viewing your data

Before we continue, you might find it useful to know about functions in the ospybook module that will help you visualize your data without opening it in another software program. These don't allow the level of interaction with the data that a GIS does, so opening it in QGIS is still a much better option for exploring the data in any depth.

VIEWING ATTRIBUTES

You can print out attribute values to your screen using the `print_attributes` function, which looks like this:

```
print_attributes(lyr_or_fn, [n], [fields], [geom], [reset])
```

- `lyr_or_fn` is either a layer object or the path to a data source. If it's a data source, the first layer will be used.
- `n` is an optional number of records to print. The default is to print them all.
- `fields` is an optional list of attribute fields to include in the printout. The default is to include them all.
- `geom` is an optional Boolean flag indicating whether the geometry type is printed. The default is `True`.
- `reset` is an optional Boolean flag indicating whether the layer should be reset to the first record before printing. The default is `True`.

For example, to print out the name and population for the first three cities in the populated places shapefile, you could do something like this from a Python interactive window:

```
>>> import ospybook as pb
>>> fn = r'D:\osgeopy-data\global\ne_50m_populated_places.shp'
>>> pb.print_attributes(fn, 3, ['NAME', 'POP_MAX'])
```

```
FID     Geometry                  NAME           POP_MAX
0       POINT (32.533, 0.583)     Bombo          75000
1       POINT (30.275, 0.671)     Fort Portal    42670
2       POINT (15.799, 40.642)    Potenza        69060
3 of 1249 features
```

Normally, you must provide arguments to functions in the order they're listed, but if you want to provide an optional argument without specifying values for earlier optional parameters, you can use keywords to specify which parameter you mean. For example, If you wanted to set `geom` to `False` without specifying a list of fields, you could do it like this:

```
pb.print_attributes(fn, 3, geom=False)
```

This function works well for viewing small numbers of attributes, but you'll probably regret using it to print all attributes of a large file.

PLOTTING SPATIAL DATA

The ospybook module also contains convenience classes to help you visualize your data spatially, although you'll learn how to do it yourself in the last chapter. To use these, you must have the matplotlib Python module installed. To plot your data, you need to create a new instance of the `VectorPlotter` class and pass a Boolean parameter to the constructor indicating if you want to use interactive mode. If interactive, the data will be drawn immediately when you plot it. If not interactive, you'll need to call `draw` after plotting the data, and everything will be drawn at once. Either way, once you've created this object, you can use it to plot your data with the `plot` method:

```
plot(self, geom_or_lyr, [symbol], [name], [kwargs])
```

- `geom_or_lyr` is a geometry, layer, or path to a data source. If a data source, the first layer will be drawn.
- `symbol` is an optional `pyplot` symbol to draw the geometries with.
- `name` is an optional name to assign to the data so it can be accessed later.
- `kwargs` are optional `pyplot` drawing parameters that are specified by keyword (you'll see the abbreviation `kwargs` used often for an indeterminate number of keyword arguments).

The `plot` function can optionally use parameters from the `pyplot` interface in matplotlib. You'll see a few used in this book, but to see more you can read the `pyplot` documentation at http://matplotlib.org/1.5.0/api/pyplot_summary.html. Let's start with an example that plots the populated places shapefile on top of country outlines:

```
>>> import os
>>> os.chdir(r'D:\osgeopy-data\global')
>>> from ospybook.vectorplotter import VectorPlotter
>>> vp = VectorPlotter(True)
>>> vp.plot('ne_50m_admin_0_countries.shp', fill=False)
>>> vp.plot('ne_50m_populated_places.shp', 'bo')
```

The first thing you do is use the built-in os module to change your working directory, which allows you to use filenames later instead of typing the entire path. Then you pass

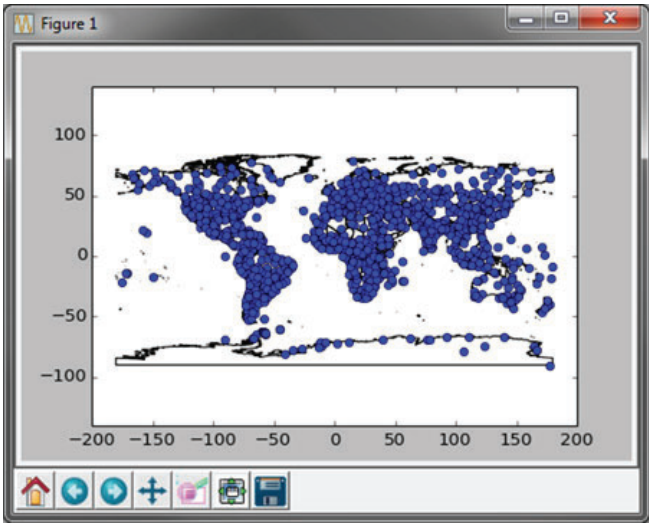


Figure 3.13 The output from plotting the global populated places shapefile on top of the country outlines

`True` to `VectorPlotter` to create an interactive plotter. The `fill` `pyplot` parameter causes the countries shapefile to be drawn as hollow polygons, and the `'bo'` symbol for populated places means blue circles. This results in a plot that looks like figure 3.13.

You don't need to do anything special if you want to use this in a script, but you should know that when the plotter isn't created with interactive mode, it will stop script execution until you close the window that pops up. I've also discovered that depending on the environment I'm running the script from, sometimes it closes itself automatically if I created it with interactive mode, so I never get the chance to view it. Because of this, if I'm using a `VectorPlotter` in a script instead of a Python interactive window, I usually create it using non-interactive mode and call `draw` at the end of the script. The source code for this chapter has examples of this.

3.4 Getting metadata about the data

Sometimes you also need to know general information about a dataset, such as the number of features, spatial extent, geometry type, spatial reference system, or the names and types of attribute fields. For example, say you want to display your data on top of a Google map. You need to make sure that your data use the same spatial reference system as Google, and you need to know the spatial extent so that you can have your map zoom to the correct part of the world. Because different geometry types have different drawing options, you also need to know geometry types to define the symbology for your features.

You've already seen how to get some of these, such as the number of features in a layer with `GetFeatureCount`. Remember that this applies to the layer and not the data source, because each layer in a data source can have a different number of features, geometry type, spatial extent, or attributes.

The spatial extent of a layer is the rectangle constructed from the minimum and maximum bounding coordinates in all directions. Figure 3.14 shows the Washington large_cities file and its extent. You can get these bounding coordinates from a layer

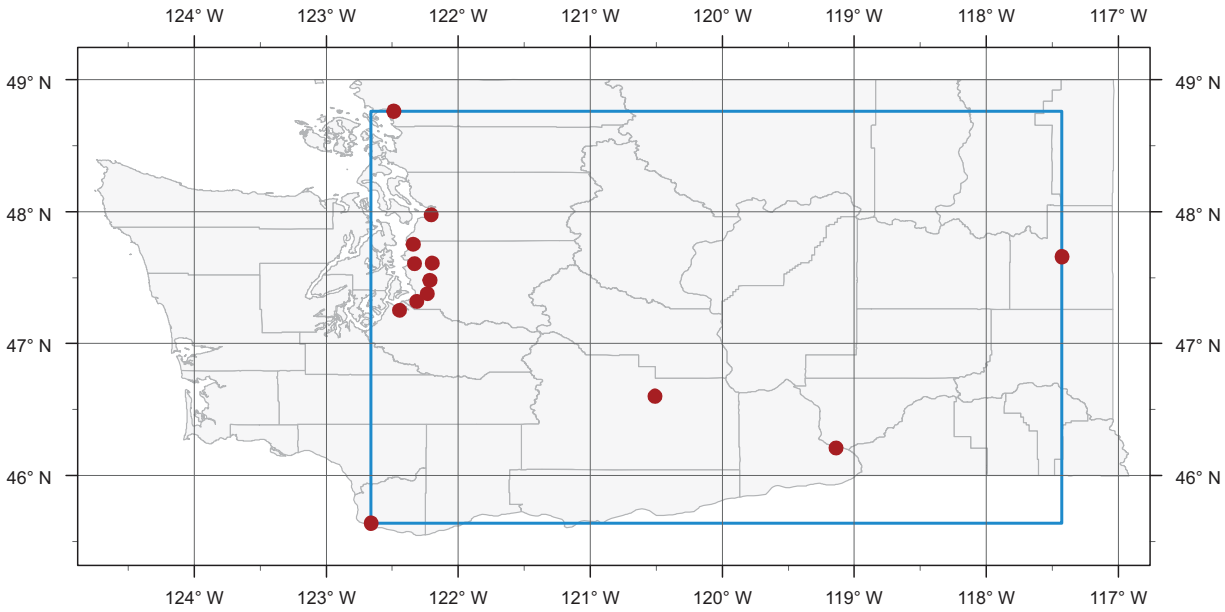


Figure 3.14 Here you can see the spatial extent of the large_cities dataset. The minimum and maximum longitude (x) values are approximately -122.7 and -117.4, respectively. The minimum and maximum latitude (y) values are approximately 45.6 and 48.8.

object with the `GetExtent` function, which returns a tuple of numbers as `(min_x, max_x, min_y, max_y)`. Here's an example:

```
>>> ds = ogr.Open(r'D:\osgeopy-data\Washington\large_cities.geojson')
>>> lyr = ds.GetLayer(0)
>>> extent = lyr.GetExtent()
>>> print(extent)
(-122.66148376464844, -117.4260482788086, 45.638729095458984,
➥ 48.759552001953125)
```

Compare these numbers to those in figure 3.14 to better understand what's returned in the extent tuple.

You can also get the geometry type from the layer object, but there's a catch. The `GetGeomType` function returns an integer instead of a human-readable string. But how is that useful? The OGR module has a number of constants, shown in table 3.1, which are basically unchangeable variables with descriptive names and numeric values. You can compare the value you get with `GetGeomType` to one of these constants in order to check if it's that geometry type. For example, the constant for point geometries is `wkbPoint` and the one for polygons is `wkbPolygon`, so continuing with the previous example, you could find out if large_cities.shp is a point or polygon shapefile like this:

```
>>> print(lyr.GetGeomType())
1
>>> print(lyr.GetGeomType() == ogr.wkbPoint)
True
>>> print(lyr.GetGeomType() == ogr.wkbPolygon)
False
```

Returns a number

Compare with constants to get type

Table 3.1 Common geometry type constants. You can find more in appendix B.

Geometry type	OGR constant
Point	`wkbPoint`
Mulitpoint	`wkbMultiPoint`
Line	`wkbLineString`
Multiline	`wkbMultiLineString`
Polygon	`wkbPolygon`
Multipolygon	`wkbMultiPolygon`
Unknown geometry type	`wkbUnknown`
No geometry	`wkbNone`

If the layer has geometries of varying types, such as a mixture of points and polygons, `GetGeomType` will return `wkbUnknown`.

> **NOTE** The wkb prefix on the OGR geometry constants stands for well-known binary (WKB), which is a standard binary representation used to exchange geometries between different software packages. Because it's binary, it isn't human-readable, but a well-known text (WKT) format does exist that is readable.

Sometimes you'd rather have a human-readable string, however, and you can get this from one of the feature geometries. The following example grabs the first feature in the layer, gets the geometry object from that feature, and then prints the name of the geometry:

```
>>> feat = lyr.GetFeature(0)
>>> print(feat.geometry().GetGeometryName())
POINT
```

Another useful piece of data you can get from the layer object is the spatial reference system, which describes the coordinate system that the dataset uses. Your GPS unit probably shows unprojected, or geographic, coordinates by default. These are the latitude and longitude coordinates that we're all familiar with. These geographic coordinates can be converted to many other types of coordinate systems, however, and if you don't know which of these systems a dataset uses, then you have no way of knowing where on the earth the coordinates refer to. Obviously, this is a crucial bit of metadata, and I'll talk more about it in chapter 8. For now, you only need to know that you can get this information. If you print it out, you'll get a string that describes the reference system in WKT format, like that shown in listing 3.2.

Listing 3.2 Example of well-known text representation of a spatial reference system

```
>>> print(lyr.GetSpatialRef())
GEOGCS["NAD83",
    DATUM["North_American_Datum_1983",
        SPHEROID["GRS 1980",6378137,298.257222101,
            AUTHORITY["EPSG","7019"]],
```

```
    TOWGS84[0,0,0,0,0,0,0],
    AUTHORITY["EPSG","6269"]],
PRIMEM["Greenwich",0,
    AUTHORITY["EPSG","8901"]],
UNIT["degree",0.0174532925199433,
    AUTHORITY["EPSG","9122"]],
AUTHORITY["EPSG","4269"]]
```

Depending on your GIS experience, this output may or may not mean much to you. Don't worry if it makes no sense now, because you'll learn all about it later.

Last, you can also get information about the attribute fields attached to the layer. The easiest way to do this is to use the schema property on the layer object to get a list of `FieldDefn` objects. Each of these contains information such as the attribute column name and data type. Here's an example of printing out the name and data type of each field:

```
>>> for field in lyr.schema:
...   print(field.name, field.GetTypeName())
...
CITIESX020 Integer
FEATURE String
NAME String
<snip>
```

Part of this output was left out in the interest of space, but you can run the code yourself to see the rest of the fields in the layer. You'll learn more about working with `FieldDefn` objects in section 3.5.2.

3.5 Writing vector data

Reading data is definitely useful, but you'll probably need to edit existing or create new datasets. Listing 3.3 shows how to create a new shapefile that contains only the features corresponding to capital cities in the global populated places shapefile. The output will look like the cities in figure 3.15.

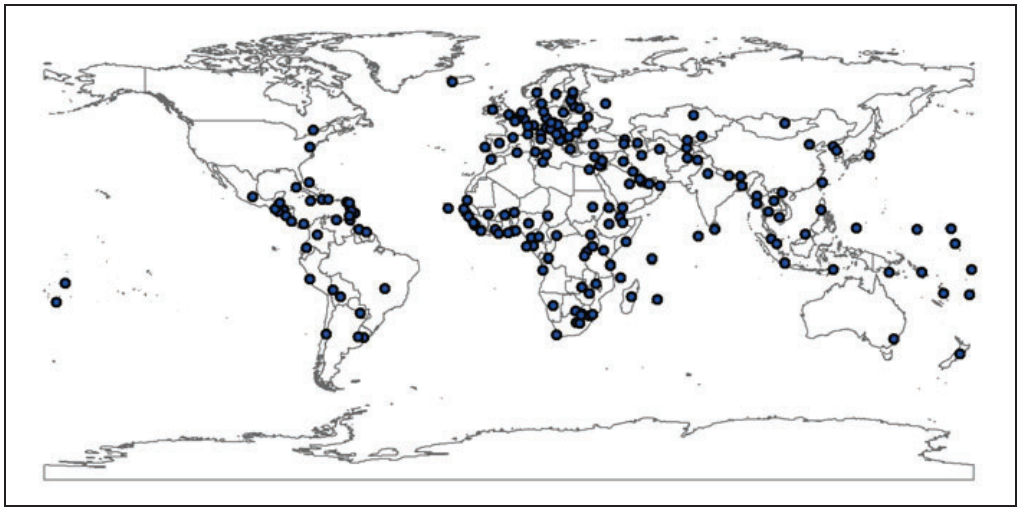


Figure 3.15 Capital cities with country outlines for reference

Listing 3.3 Exporting capital cities to a new shapefile

```
import sys
from osgeo import ogr

ds = ogr.Open(r'D:\osgeopy-data\global', 1)
if ds is None:
    sys.exit('Could not open folder.')
in_lyr = ds.GetLayer('ne_50m_populated_places')

if ds.GetLayer('capital_cities'):
    ds.DeleteLayer('capital_cities')
out_lyr = ds.CreateLayer('capital_cities',
                         in_lyr.GetSpatialRef(),
                         ogr.wkbPoint)
out_lyr.CreateFields(in_lyr.schema)

out_defn = out_lyr.GetLayerDefn()
out_feat = ogr.Feature(out_defn)
for in_feat in in_lyr:
    if in_feat.GetField('FEATURECLA') == 'Admin-0 capital':
        geom = in_feat.geometry()
        out_feat.SetGeometry(geom)
        for i in range(in_feat.GetFieldCount()):
            value = in_feat.GetField(i)
            out_feat.SetField(i, value)
        out_lyr.CreateFeature(out_feat)

del ds
```

Open folder data source for writing

Get input shapefile

Delete layer if it exists

Create a point layer

Create a blank feature

Copy geometry and attributes

Insert the feature

Close files

In this example you open up a folder instead of a shapefile as the data source. A nice feature of the shapefile driver is that it will treat a folder as a data source if a majority of the files in the folder are shapefiles, and each shapefile is treated as a layer. Notice that you pass `1` as the second parameter to `Open`, which will allow you to create a new layer (shapefile) in the folder. You pass the shapefile name, without the extension, to `GetLayer` to get the populated places shapefile as a layer. Even though you open it differently here than in listing 3.1, you can use it in exactly the same way.

Because OGR won't overwrite existing layers, you check to see if the output layer already exists, and delete it if it did. Obviously you wouldn't want to do this if you didn't want the layer overwritten, but in this case you can overwrite data as you test different things.

Then you create a new layer to store your output data in. The only required parameter for `CreateLayer` is a name for the layer, which should be unique within the data source. You do have, however, several optional parameters that you should set when possible:

```
CreateLayer(name, [srs], [geom_type], [options])
```

- `name` is the name of the layer to create.
- `srs` is the spatial reference system that the layer will use. The default is `None`, meaning that no spatial reference system will be assigned.

- `geom_type` is a geometry type constant from table 3.1 that specifies the type of geometry the layer will hold. The default is `wkbUnknown`.
- `options` is an optional list of layer-creation options, which only applies to certain vector format types.

The first of these optional parameters is the spatial reference, which defaults to `None` if not provided. Remember that without spatial reference information, it's extremely difficult to figure out where the features are on the planet. Sometimes the spatial reference is implicit in the data; for example, KML only supports unprojected coordinates using the WGS 84 datum, but you should set this if possible. In this case, you copy the spatial reference information from the original shapefile to the new one. We'll discuss spatial reference systems and how to use them in more detail in chapter 8.

The second optional parameter is one of the OGR geometry type constants from either table 3.1 or appendix B. This specifies the type of geometries that the layer will contain. If not provided, it defaults to `ogr.wkbUnknown`, although in many cases this will be updated to the correct value after you add features to the layer and it can be determined from them.

The last optional parameter is a list of layer-creation option strings in the form of *option=value*. These are documented for each driver on the OGR formats webpage. Not all vector data formats have layer-creation options, and even if a format does have options, you're under no obligation to use them.

You use the following code to create a new point shapefile called capital_cities.shp that uses the same spatial reference system as the populated places shapefile. You do one more thing, though. The `schema` property on the input layer returns a list of attribute field definitions for that layer, and you pass that list to `CreateFields` to create the exact same set of attribute fields in the new layer:

```
out_lyr = ds.CreateLayer('capital_cities',
                         in_lyr.GetSpatialRef(),
                         ogr.wkbPoint)
out_lyr.CreateFields(in_lyr.schema)
```

Now, to add a feature to a layer, you need to create a dummy feature that you add the geometry and attributes to, and then you insert that into the layer. The next step is to create this blank feature. Creating a feature requires a feature definition that contains information about the geometry type and all of the attribute fields, and this is used to create an empty feature with the same fields and geometry type. You need to get the feature definition from the layer you plan to add features to, but you must do it after you've added, deleted, or updated any fields. If you get the feature definition first, and then change the fields in any way, the definition will be out of date. This means that a feature you try to insert based on this outdated definition will not match reality, as seen in figure 3.16. This will cause Python to die a horrible death, and you definitely don't want that.

```
out_defn = out_lyr.GetLayerDefn()
out_feat = ogr.Feature(out_defn)
```

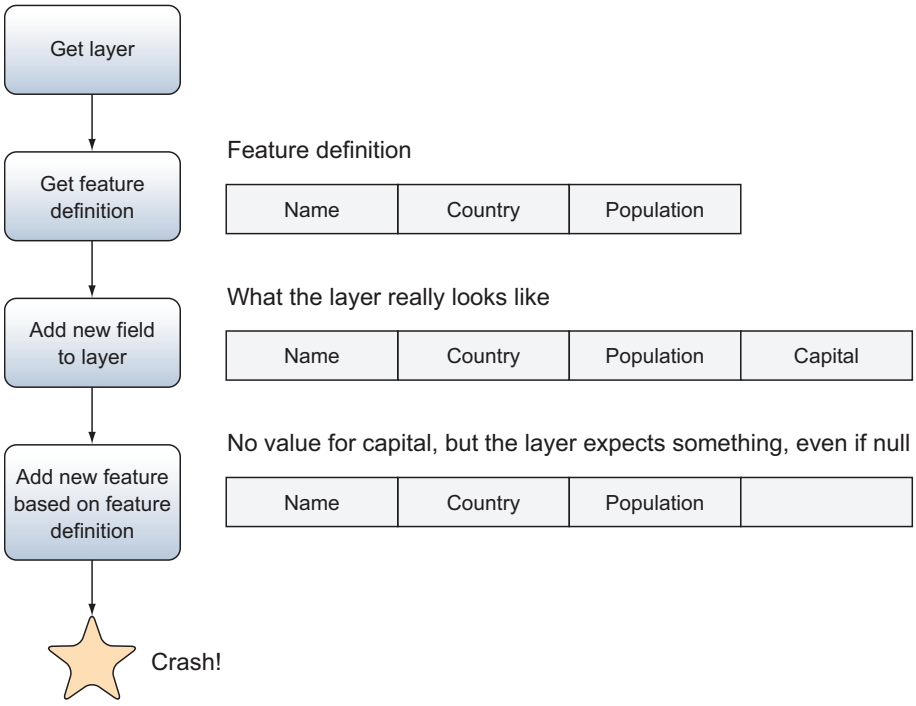


Figure 3.16 Always get feature definitions after making changes to fields, or the definition will not match reality.

Now that you have a feature to put information into, it's time to start looping through the input dataset. For each feature, you check to see if its `FEATURECLA` attribute was equal to `'Admin-0 capital'`, which means it's a capital city. If it is, then you copy the geometry from it into the dummy feature. Then you loop through all of the fields in the attribute table and copy the values from the input feature into the output feature. This works because you create the fields in the new shapefile based on the fields in the original, so they're in the same order in both shapefiles. If they were in different orders, you'd have to use their names to access them, but you can use indexes here because you know that they match:

```
for in_feat in in_lyr:
    if in_feat.GetField('FEATURECLA') == 'Admin-0 capital':
        geom = in_feat.geometry()
        out_feat.SetGeometry(geom)
        for i in range(in_feat.GetFieldCount()):
            value = in_feat.GetField(i)
            out_feat.SetField(i, value)
        out_lyr.CreateFeature(out_feat)
```

Once you copy all of the attribute fields over, you insert the feature into the layer using `CreateFeature`. This function saves a copy of the feature, including all of the information you add to it, to the layer. The feature object can then be reused, and whatever you do to it won't affect the data that have already been added to the layer.

This way you don't have the overhead of creating multiple features, because you can create a single one and keep editing its data each time you want to add a new feature to the layer.

You delete the `ds` variable at the end of the script, which forces the files to close and all of your edits to be written to disk. Deleting the layer variable doesn't do the trick; you must close the data source. If you wanted to keep the data source open, you could call `SyncToDisk` on either the layer or data source object instead, like this:

```
ds.SyncToDisk()
```

WARNING You must close your files or call `SyncToDisk` to flush your edits to disk. If you don't do this, and your interactive environment still has your data source open, you'll be disappointed to find an empty dataset.

It's always a good idea to carefully inspect your output to make sure you get the results you want. The best way would be to open it in QGIS, or you could get a good idea by plotting it from Python (figure 3.17):

```
>>> vp = VectorPlotter(True)
>>> vp.plot('ne_50m_admin_0_countries.shp', fill=False)
>>> vp.plot('capital_cities.shp', 'bo')
```

Let's return to the topic of adding attribute values for a moment. You might be wondering if multiple functions exist for setting attribute field values as with retrieving values. The answer is generally no. Most data will be converted to the correct type for you, but you may not like the results if a conversion isn't possible. For example, pretend for a minute that you made a mistake and inserted the population into the `Name` field, and the name into the `Population` field. Do you think that the population could be converted to a string and successfully inserted into the `Name` field? How about converting the country name to a number so it could go in the `Population` field?

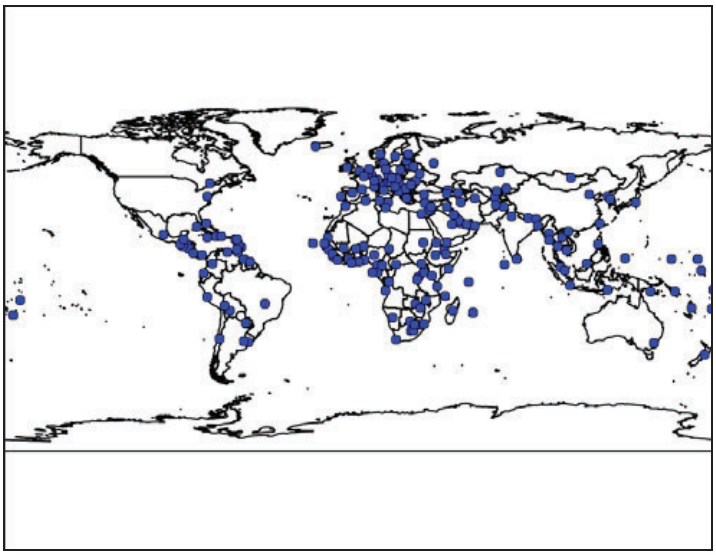


Figure 3.17 The result of plotting the new capital cities shapefile on top of country outlines

Well, converting a number to a string works fine, but converting a string to a number is problematic. The string `"3578"` can be translated into the number 3578, but what about the string `"Russia"`? If you try it in a Python interactive window by typing `int('Russia')`, you'll get an error, but OGR will insert a zero into the `Population` field instead of crashing. Sometimes this behavior is to your advantage because you don't need to convert data before inserting it in a feature, but it can also be a problem if you mistakenly try to insert the wrong type of data into a field.

3.5.1 Creating new data sources

You used an existing data source in listing 3.3, but sometimes you'll need to create new ones. Fortunately, it's not difficult. Perhaps the most important part is that you use the correct driver. It's the driver that does the work here, and each driver only knows how to work with one vector format, so using the correct one is essential. For example, the GeoJSON driver won't create a shapefile, even if you ask it to create a file with an .shp extension. As shown in figure 3.18, the output will have an .shp extension, but it will still be a GeoJSON file at heart.

You have a couple of ways to get the required driver. The first is to get the driver from a dataset that you've already opened, which will allow you to create a new data source using the same vector data format as the existing data source. In this example, the `driver` variable will hold the ESRI shapefile driver:

```
ds = ogr.Open(r'D:\osgeopy-data\global\ne_50m_admin_0_countries.shp')
driver = ds.GetDriver()
```

The second way to get a driver object is to use the OGR function `GetDriverByName` and pass it the short name of the driver. Remember that these names are available on the OGR website, by using the `ogrinfo` utility that comes with GDAL/OGR, or the `print_drivers` function available in the code accompanying this book. This example will get the GeoJSON driver:

```
json_driver = ogr.GetDriverByName('GeoJSON')
```

Once you have a driver object, you can use it to create an empty data source by providing the data source name. This new data source is automatically open for writing, and

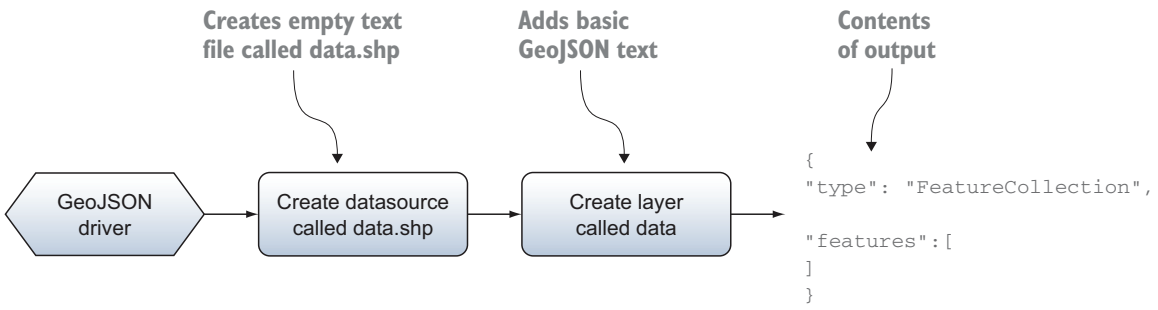


Figure 3.18 Using the GeoJSON driver to create a file with an .shp extension will still create a GeoJSON file, not a shapefile.

you can add layers to it the way you did in listing 3.3. If the data source can't be created, then `CreateDataSource` returns `None`, so you need to check for this condition:

```
json_ds = json_driver.CreateDataSource(json_fn)
if json_ds is None:
    sys.exit('Could not create {0}.'.format(json_fn))
```

A few data formats have creation options that you can use when creating a data source, although these aren't required. Like layer-creation options, these parameters are documented on the OGR website. Don't confuse the two, because data source and layer-creation options are two different things. Both types are passed as a list of strings, however. Let's see how you'd use a data source–creation option to create a full-fledged SpatiaLite data source instead of SQLite. This will fail if your version of OGR wasn't built with SpatiaLite support, though:

```
driver = ogr.GetDriverByName('SQLite')
ds = driver.CreateDataSource(r'D:\ osgeopy-data\global\earth.sqlite',
                             ['SPATIALITE=yes'])
```

Another thing to be aware of when creating new data sources is that you can't overwrite an existing data source. If a chance exists that your code might legitimately try to overwrite a dataset, then you'll need to delete the old one before attempting to create the new one. One way to deal with this would be to use the Python `os.path.exists` function to see if a file already exists before you attempt to create a data source; or you could wait and deal with it if your original attempt fails, either after checking for `None` or by using a try/except block. Either way, you should use the driver to delete the existing source instead of using a Python built-in function. Why? Because the driver will make sure that all required files are deleted. For example, if you're deleting a shapefile, the shapefile driver will delete the .shp, .dbf, .shx, and any other optional files that may be present. If you were using the Python built-in module to delete the shapefile, you'd have to make sure your code checked for all of these files. Here's an example of one way to deal with an existing data source:

```
if os.path.exists(json_fn):
    json_driver.DeleteDataSource(json_fn)
json_ds = json_driver.CreateDataSource(json_fn)
if json_ds is None:
    sys.exit('Could not create {0}.'.format(json_fn))
```

TIP If you try to create a shapefile as a data source rather than a layer (where the data source is the containing folder), and the shapefile already exists, you'll get an odd error message saying that the shapefile isn't a directory.

Using OGR exceptions

By default, OGR doesn't raise an error if it has a problem, such as failing to create a new data source. This is why you check for `None`, but Python programmers generally expect an error to be raised instead. You can enable this behavior if you'd like, by

calling `ogr.UseExceptions()` at the beginning of your code. Although most of the time this works as anticipated, I've discovered that it doesn't always raise an error when I expect. For example, no error is raised if OGR fails to open a data source. However, in instances where it does raise an error, you don't need to check for `None` before continuing. Using exceptions also gives you flexibility with handling errors.

For example, here's a contrived situation where I'm pretending to process data, then I want to save some temporary data to a GeoJSON file, and then I want to keep processing something else. If I can't create the temporary file, I want to skip that step and go on to the next bit of data processing rather than crashing. Here's the sample code:

```
ogr.UseExceptions()
fn = r'D:\ osgeopy-data\global\africa.geojson'
driver = ogr.GetDriverByName('GeoJSON')
print('Doing some preliminary analysis...')

try:
    ds = driver.CreateDataSource(fn)
    lyr = ds.CreateLayer('layer')
    # Do more stuff, like create fields and save data

except RuntimeError as e:
    print(e)

print('Doing some more analysis...')
```

Turn on exceptions

Attempt to save some data

Print error message and continue

Suppose that the africa.geojson file already exists. This code doesn't check for that, so you know it will fail when you call `CreateDataSource`. If you weren't using OGR exceptions, this script would fail at that point and never get to the last `print` statement. But because you're using exceptions, you'll get an error message saying that the file couldn't be created, and then it will continue on to the last `print` statement, and the output will look like this:

```
Doing some preliminary analysis...
The GeoJSON driver does not overwrite existing files.
Doing some more analysis...
```

Try it out yourself and comment out the first line, and watch how the behavior changes.

3.5.2 Creating new fields

You saw in listing 3.3 how to copy attribute field definitions from one layer to another, but you can also define your own custom fields. Several different field types are available, but not all are supported by all data formats. This is another situation when the online documentation for the various formats will come in handy, so hopefully you've bookmarked that page.

To add a field to a layer, you need a `FieldDefn` object that contains the important information about the field, such as name, data type, width, and precision. The `schema` property you used in listing 3.3 returns a list of these, one for each field in the

layer. You can create your own, however, by providing the name and data type for the new field to the `FieldDefn` constructor. The data type is one of the constants from table 3.2.

Table 3.2 Field type constants. There are more shown in appendix B, but I have been unable to make them work in Python.

Field data type	OGR constant
Integer	`OFTInteger`
List of integers	`OFTIntegerList`
Floating point number	`OFTReal`
List of floating point numbers	`OFTRealList`
String	`OFTString`
List of strings	`OFTStringList`
Date	`OFTDate`
Time of day	`OFTTime`
Date and time	`OFTDateTime`

After you create a basic field definition, but before you use it to add a field to the layer, you can add other constraints such as floating-point precision or field width, although I've noticed that these don't always have an effect, depending on the driver being used. For example, I haven't been able to set a precision in a GeoJSON file, and I've also discovered that you must set a field width if you want to set field precision in a shapefile. This example would create two fields to hold x and y coordinates with a precision of `3`:

```
coord_fld = ogr.FieldDefn('X', ogr.OFTReal)
coord_fld.SetWidth(8)
coord_fld.SetPrecision(3)
out_lyr.CreateField(coord_fld)
coord_fld.SetName('Y')
out_lyr.CreateField(coord_fld)
```

Create and add the first field

Reuse the FieldDefn to create a second field

You might have noticed that you don't create two different field definition objects here. Once you've used the field definition to create a field in the layer, you can change the definition's attributes and reuse it to create another field, which makes this easier because you want two fields that were identical except in name.

Also, sometimes the field width will be ignored if it's too small for the data provided. For example, if you create a string field with a width of `6`, but then try to insert a value that's 11 characters long, in certain cases the width of the field would increase to hold the entire string. This isn't always possible, however, and it's best to be specific about what you want rather than hope something like this will conveniently happen.

3.6 Updating existing data

Sometimes you need to update existing data rather than create an entirely new dataset. Whether this is possible, and which edits are supported, depends on the format of the data. For example, you can't edit GeoJSON files, but many different edits are allowed on shapefiles. We'll discuss getting information about what's supported in the next chapter.

3.6.1 Changing the layer definition

Depending on the type of data you're working with, you can edit the layer definition by adding new fields, deleting existing ones, or changing field properties such as name. As with adding new fields, you need a field definition to change a field. Once you have a field definition that you're happy with, you use the `AlterFieldDefn` function to replace the existing field with the new one:

```
AlterFieldDefn(iField, field_def, nFlags)
```

- `iField` is the index of the field you want to change. A field name won't work in this case.
- `field_def` is the new field definition object.
- `nFlags` is an integer that is the sum of one or more of the constants shown in table 3.3.

Table 3.3 Flags used to specify which properties of a field definition can be changed. To use more than one, simply add them together

Field properties that need to change	OGR constant
Field name only	`ALTER_NAME_FLAG`
Field type only	`ALTER_TYPE_FLAG`
Field width and/or precision only	`ALTER_WIDTH_PRECISION_FLAG`
All of the above	`ALTER_ALL_FLAG`

To change a field's properties, you need to create a field definition containing the new properties, find the index of the existing field, and decide which constants from table 3.3 to use to ensure your changes take effect. To change the name of a field from `'Name'` to `'City_Name'`, you might do something like this:

```
i = lyr.GetLayerDefn().GetFieldIndex('Name')
fld_defn = ogr.FieldDefn('City_Name', ogr.OFTString)
lyr.AlterFieldDefn(i, fld_defn, ogr.ALTER_NAME_FLAG)
```

If you needed to change multiple properties, such as both the name and the precision of a floating-point attribute field, you'd pass the sum of `ALTER_NAME_FLAG` and `ALTER_WIDTH_PRECISION_FLAG`, like this:

```
lyr_defn = lyr.GetLayerDefn()
i = lyr_defn.GetFieldIndex('X')
```

```
width = lyr_defn.GetFieldDefn(i).GetWidth()
fld_defn = ogr.FieldDefn('X_coord', ogr.OFTReal)
fld_defn.SetWidth(width)
fld_defn.SetPrecision(4)
flag = ogr.ALTER_NAME_FLAG + ogr.ALTER_WIDTH_PRECISION_FLAG
lyr.AlterFieldDefn(i, fld_defn, flag)
```

Notice that you use the original field width when creating the new field definition. I found out the hard way that if you don't set the width large enough to hold the original data, then the results will be incorrect. To get around the problem, use the original width. For the precision change to take effect, all records must be rewritten. Making the precision larger than it was won't give you more precision, however, because data can't be created from thin air. The precision can be decreased, however.

Instead of summing up flag values, you could cheat and just use `ALTER_ALL_FLAG`. Only do this if your new field definition is exactly what you want the field to look like after editing, however. The other flags limit what can change, but this one doesn't. For example, if your field definition has a different data type than the original field but you pass `ALTER_NAME_FLAG`, then the data type will not change, but it will if you pass `ALTER_ALL_FLAG`.

3.6.2 Adding, updating, and deleting features

Adding new features to existing layers is exactly the same as adding them to brand-new layers. Create an empty feature based on the layer definition, populate it, and insert it into the layer. Updating features is much the same, except you work with features that already exist in the layer instead of blank ones. Find the feature you want to edit, make the desired changes, and then update the information in the layer by passing the updated feature to `SetFeature` instead of `CreateFeature`. For example, you could do something like this to add a unique ID value to each feature in a layer:

```
lyr.CreateField(ogr.FieldDefn('ID', ogr.OFTInteger))
n = 1
for feat in lyr:
    feat.SetField('ID', n)
    lyr.SetFeature(feat)
    n += 1
```

First you add an ID field, and then you iterate through the features and set the ID equal to the value of the `n` variable. Because you increment `n` each time through the loop, each feature has a unique ID value. Last, you update the feature in the layer by passing it to `SetFeature`.

Deleting features is even easier. All you need to know is the FID of the feature you want to get rid of. If you don't know that number off the top of your head, or through another means, you can get it from the feature itself, like this:

```
for feat in lyr:
    if feat.GetField('City_Name') == 'Seattle':
        lyr.DeleteFeature(feat.GetFID())
```

FID	Name
0	Lima
1	Paris
Deleted	Toronto
3	Geneva
4	Oslo
5	Moscow
Deleted	Tokyo
7	Cape Town

Vacuum/repack

FID	Name
0	Lima
1	Paris
2	Geneva
3	Oslo
4	Moscow
5	Cape Town

Figure 3.19 The effect of vacuuming or repacking a database. Notice that the FID values change.

For each feature in the layer, you check to see if its `'City_Name'` attribute is equal to `'Seattle'`, and if it is, you retrieve the FID from the feature itself and then pass that number to `DeleteFeature`.

Certain formats don't completely kill the feature at this point, however. You may not see it, but sometimes the feature has only been marked for deletion instead of totally thrown out, so it's still lurking in the shadows. Because of this, you won't see any other features get assigned that FID, and it also means that if you've deleted many features, there may be a lot of needlessly used space in your file. See figure 3.19 for a simple example. Deleting these features will reclaim this space. If you have much experience with relational databases, you should be familiar with this idea. It's similar to running Compact and Repair on a Microsoft Access database or using `VACUUM` on a PostgreSQL database.

How to go about reclaiming this space, or determining if it needs to be done, is dependent on the vector data format being used. Here are examples for doing it for shapefiles and SQLite:

```
ds.ExecuteSQL('REPACK ' + lyr.GetName())
ds.ExecuteSQL('VACUUM')
```

Shapefile

SQLite

In both cases, you need to open the data source and then execute a SQL statement on it that compacts the database. For shapefiles you need to know the name of the layer, so if the layer is called `"cities"`, then the SQL would be `"REPACK cities"`.

Another issue with shapefiles is that they don't update their metadata for spatial extent when existing features are modified or deleted. If you edit existing geometries or delete features, you can ensure that the spatial extent gets updated by calling this:

```
ds.ExecuteSQL('RECOMPUTE EXTENT ON ' + lyr.GetName())
```

This isn't necessary if you insert features, however, because those extent changes are tracked. It's also not necessary if there's no chance that your edits change the layer's extent.

3.7 Summary

- Vector data formats are most appropriate for features that can be characterized as a point, line, or polygon.
- Each geographic feature in a vector dataset can have attribute data, such as name or population, attached to it.
- The type of geometry used to model a given feature may change depending on scale. A city could be represented as a point on a map of an entire country, but as a polygon on a map of a smaller area, such as a county.
- Vector datasets excel for measuring relationships between geographic features such as distances or overlaps.
- You can use OGR to read and write many different types of vector data, but which ones depend on which drivers have been compiled into your version of GDAL/OGR.
- Data sources can contain one or more layers (depending on data format), and in turn, layers can contain one or more features. Each feature has a geometry and a variable number of attribute fields.
- Newly created data sources are automatically opened for writing. If you want to edit existing data, remember to open the data source for writing.
- Remember to make changes to the layer, such as adding or deleting fields, before getting the layer definition and creating a feature for adding or updating data.

4 Working with different vector file formats

This chapter covers

- Choosing a vector data file format
- Working with various vector data formats
- Checking what edits are allowed on a data source

As mentioned in the previous chapter, there are many different vector file formats, and they're not always interchangeable, at least in a practical sense. Certain formats are more appropriate for certain uses than others. In this chapter you'll learn several of the differences and their strengths and weaknesses.

Another consideration with format is what you can and can't do with the data using OGR. In general, working with one type is the same as working with another, but sometimes how you open the data source is different. The larger issue is the difference in capabilities of each driver. For example, certain formats can be read from but not written to, and others can be created but existing data can't be edited. You'll also learn how to determine what you can and can't do with a dataset.

4.1 Vector file formats

Up to this point, you've only worked with shapefiles, but many more vector file formats are available. Chances are that you'll probably only use a handful of them on a regular basis, but you need to have an idea of the available options. Several formats have open specifications and are supported by many different software programs, while others are used more sparingly. Certain formats also support more capabilities than others. Most of these formats allow for easy transfer from one user to another, much like you can give someone else your spreadsheet file. A few use database servers, however, which allows for many users to access and edit the same dataset at a central location, but sometimes makes it more difficult to move the data from one place to another.

4.1.1 File-based formats such as shapefiles and geoJSON

What I call *file-based formats* are made up of one or more files that live on a disk drive and can be easily transferred from one location to another, such as from your hard drive to another computer or an external drive. Several of these are relational databases, but are designed to be easily moved around (think of Microsoft Access relational databases), so they're considered file-based for the purposes of this discussion. Several of these formats have open standards so anyone can create software to use them, while others are proprietary and limited to smaller numbers of software. Examples of open formats are GeoJSON, KML, GML, shapefiles, and SpatiaLite.

Spatial data can also be stored in Excel spreadsheets, comma- or tab-delimited files, or other similar formats, although this is most common for point data when only x and y coordinates are required. Most spatial data, however, is stored using formats designed specifically for GIS data. Several of these formats are plain text, meaning that you can open them in any text editor and look at them, and others are binary files that require software capable of understanding them.

As mentioned previously, one advantage of plain text files is that you can open them in a text editor and inspect their contents. You can even edit them by hand, rather than using GIS software, if you're so inclined. Listing 4.1 shows an example of a GeoJSON file that contains two cities in Switzerland, Geneva and Lausanne, both represented as points.

Listing 4.1 An example GeoJSON file with two features

```
{
  "type": "FeatureCollection",
  "features": [
    {
      "type": "Feature",
      "properties": { "NAME": "Geneva", "PLACE": "city" },
      "geometry": {
        "type": "Point",
        "coordinates": [ 6.1465886, 46.2017589 ]
      }
```

```
    },
    {
      "type": "Feature",
      "properties": { "NAME": "Lausanne", "PLACE": "city" },
      "geometry": {
        "type": "Point",
        "coordinates": [ 6.6327025, 46.5218269 ]
      }
    },
  ]
}
```

It's okay if you don't understand everything in this example. The point here is that you can open and edit the file in a text editor instead of using GIS software. For example, you could easily fix the spelling of a city name or tweak one of the point coordinates. While we're on the subject, it's worth mentioning that small GeoJSON files are automatically rendered as interactive maps when uploaded to GitHub. The example shown here is saved as a gist at https://gist.github.com/cgarrard/8049400. If you have a GitHub account, you can copy this gist to your own account, make changes, and instantly see the result.

Plain text formats such as GeoJSON, KML, and GML are popular for transferring small amounts of data and for web applications, but they don't work so well for data analysis. For one thing, all three of these formats allow different geometry types to be present in the same dataset, which GIS software doesn't really appreciate. For example, data in the popular shapefile format contains all points, all lines, or all polygons, but not a mixture. Therefore, a shapefile could contain roads (lines) or city boundaries (polygons), but not both. A GeoJSON file, on the other hand, can contain a combination of all three geometries in the same dataset, such as the roads and city boundaries mentioned previously that would have to live in two different shapefiles. Because you have only one file to download and process, this is an excellent solution for passing data to a web browser so it can render it on a map. However, most GIS software expects only points, only lines, or only polygons, and won't read the data correctly if it has a mixture. If you need to load the data into GIS software, don't combine multiple geometry types into one dataset, even when allowed.

Perhaps a more serious problem with plain text formats when it comes to data analysis is that they don't have the same indexing capabilities as many binary formats. Indexes are used for searching and accessing data quickly. Attribute indexes allow for searching on values in the attribute fields for the features, such as searching for all cities in a dataset with a population over 100,000. Spatial indexes store information about the spatial location of features in the dataset so that searching can be limited to features in a certain geographic area, for example, when you overlay a small watershed polygon on a larger dataset of water-monitoring stations. A spatial index would be used to quickly find the monitoring stations that fall within the watershed boundary. Both of these operations, finding large cities and finding water-monitoring stations, would be slow on large datasets if the appropriate attribute or spatial index didn't exist. In addition, spatial indexes can help a dataset be drawn more quickly

because they help find the features that fall within the viewport. For example, if you're looking at Asian cities and zoom in on Japan, the spatial index helps find Japanese cities faster while ignoring cities in western China.

These issues aren't as important with small datasets, but they're extremely important with large ones. Certain formats have ways around these problems, though. For example, although the KML format doesn't have true spatial indexes, it does allow for datasets to be broken up into different files for different spatial locations. This allows for smaller datasets to be loaded as a user zooms and pans around the map, which increases rendering speed.

Several vector data formats use familiar desktop-based, or personal, relational database software under the hood. This is true for Esri personal geodatabases and GeoMedia .mdb files, which use Microsoft Access databases to store data. Another example of a vector format based on an existing database format is SpatiaLite, a spatial extension for the SQLite database management system. These vector data formats can take advantage of the capabilities built into the database software, such as indexes. The underlying database also imposes much stricter rules for storing data. For example, all geographic features in a dataset must have the same geometry type and the same set of attribute fields. Similar to the way nonspatial databases can contain multiple tables, a spatial database can contain multiple datasets. Although an individual dataset is limited to a single geometry type, a solitary database file can contain multiple datasets, each with different geometry types and attribute fields. This is convenient for keeping related datasets together and for moving them from disk to disk. Figure 4.1 shows a schematic of a single SpatiaLite database file that contains multiple datasets with different geometries.

Other vector formats consist of several files, such as the ever popular shapefile. These datasets store geometries, attribute values, and indexes in separate files. If you move a shapefile from one location to another, you need to ensure that you move all of the required files. Other format types that require multiple files make it a bit easier by using dedicated folders that contain the necessary files. As with shapefiles, you don't need to know anything about the individual files, but you shouldn't change anything in the folder. Two examples of formats that use this system are Esri grids and file geodatabases.

Many other vector data formats haven't been mentioned here, but you should now have an idea of the types of formats and their strengths and weaknesses.

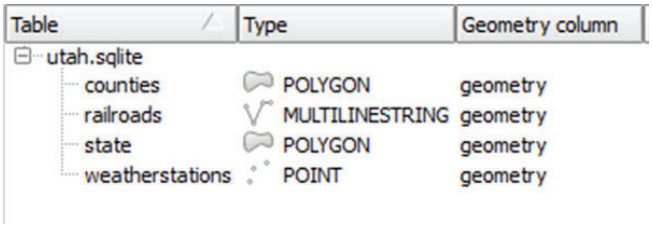


Figure 4.1 A sample SpatiaLite database containing multiple layers with different geometry types. All of these various datasets are contained within one easily transportable file.

4.1.2 Multi-user database formats such as PostGIS

You've seen that file-based formats come in many shapes and sizes, including desktop relational database models such as SpatiaLite. One limitation of these formats is that they don't allow multiple people to edit, or sometimes even use, a specific dataset at the same time. This is where the multi-user client-server database architecture comes in, because the data are stored in a database that is accessible by multiple clients across the network. Users access data from the server rather than opening a file on a local disk. Although this is certainly not for everyone, it's a great choice for making data available to many users from a central location. This is especially useful if the data are updated frequently or are used by many different users, because all users will instantly have access to the updated data. It also allows multiple people to edit a dataset at once, which isn't usually possible with file-based formats. In addition, in many cases the indexing and querying capabilities of these database systems provide faster performance when accessing data.

The most popular client-server database solutions for spatial data include PostgreSQL with the PostGIS spatial extension, ArcSDE, SQL Server, and Oracle Spatial and Graph. If you want to host the data on your own computer, you need to invest in a system like these. My favorite is PostGIS (www.postgis.net) because it's open source and provides a feature-rich environment with many functions, operators, and indexes that are specific to spatial data. Even with huge amounts of data, you can still get good performance. Although you can't zip up a PostGIS dataset and email it to a colleague, it comes with utilities to import and export several popular file-based formats, and it's straightforward to run a query and export the data to a portable format. Not only does PostGIS store the data, but you can use it for many types of analyses as well, without the need for other GIS software. PostGIS also works with raster data.

If you're not familiar with relational databases, then it might take effort to set one of these systems up and learn how to use it. But it's extremely powerful and worth the investment in brain cells if you need to give multiple users simultaneous access to data.

4.2 Working with more data formats

Until now we've only worked with one data format out of many. The basics don't change between formats, though. Once you open the data source, reading the data is pretty much the same. But for kicks, let's look at several formats that support more than one layer, because we haven't done that yet. Until now, we've used the first and only layer in a data source, but if multiple layers exist, you need to know either the name or the index of the one you're interested in. Generally, I'd use ogrinfo to get this information, but because this is a book on Python, let's write a simple function that opens a data source, loops through the layers, and prints their names and indexes:

```
def print_layers(fn):
    ds = ogr.Open(fn, 0)
    if ds is None:
        raise OSError('Could not open {}'.format(fn))
    for i in range(ds.GetLayerCount()):
        lyr = ds.GetLayer(i)
        print('{0}: {1}'.format(i, lyr.GetName()))
```

This function takes the filename of the data source as a parameter, and the first thing it does is open the file. Then it uses `GetLayerCount` to find out how many layers the data source contains, and iterates through a loop that many times. Each time through the loop, it uses the `i` variable to get the layer at the index corresponding to that iteration. Then it prints the name of the layer and its index. This function is included in the ospybook module, and you'll use it to inspect other data sources in the following examples.

4.2.1 SpatiaLite

Let's start with a SpatiaLite database. This type of data source can contain many different layers, all with unique (and hopefully descriptive) names. To see this, list the layers in the natural_earth_50m.sqlite file in the data download:

```
>>> import ospybook as pb
>>> pb.print_layers(r'D:\osgeopy-data\global\natural_earth_50m.sqlite')
0: countries
1: populated_places
```

As you can see, the dataset has two layers. How would you get a handle to the populated_places layer? Well, you could use either the index or the layer name, so both `ds.GetLayer(1)` and `ds.GetLayer('populated_places')` would do the trick. It's probably better to use the name rather than the index, however, because the index might change if other layers are added to the data source. To prove that this works, try plotting the layer, which will be dots representing cities around the world, as shown in figure 4.2.

```
>>> ds = ogr.Open(r'D:\osgeopy-data\global\natural_earth_50m.sqlite')
>>> lyr = ds.GetLayer('populated_places')
>>> vp = VectorPlotter(True)
>>> vp.plot(lyr, 'bo')
```

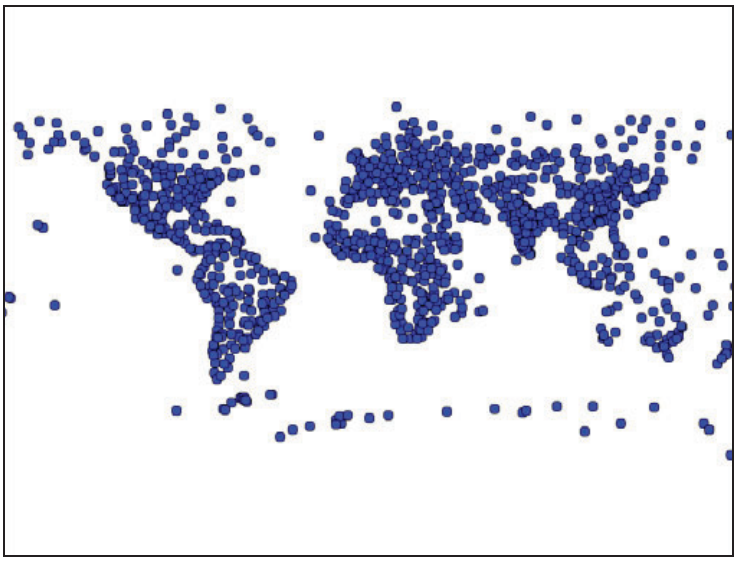


Figure 4.2 The populated_places layer in natural_earth_50m.sqlite

Ogrinfo

GDAL comes with several extremely useful command-line utilities, and in fact, you've already seen how to use ogrinfo to find out which vector data formats your version of OGR supports. You can also use ogrinfo to get information about specific data sources and layers. If you pass it a data source name, it will print a list of layers contained in that data source:

```
D:\osgeopy-data\global>ogrinfo natural_earth_50m.sqlite
INFO: Open of `natural_earth_50m.sqlite'
      using driver `SQLite' successful.
1: countries (Multi Polygon)
2: populated_places (Point)
```

You can also use ogrinfo to see metadata about a layer and even all of the attribute data. This example will show a summary only (`-so`) of the countries layer in the natural earth SQLite database. This includes metadata such as the extent, spatial reference, and a list of attribute fields and their data types. The second will show all attribute values for the first feature in the layer.

```
ogrinfo -so natural_earth_50m.sqlite countries
```

To display all of the attribute values for the feature with an FID of 1, you could do something like this, where `-q` means *don't print the metadata* and `-geom=NO` means *don't print out a text representation of the geometry* (which would be long).

```
ogrinfo -fid 1 -q -geom=NO natural_earth_50m.sqlite countries
```

See http://www.gdal.org/ogrinfo.html for full ogrinfo documentation.

4.2.2 PostGIS

What about connecting to a database server such as the PostGIS spatial extension for PostgreSQL? Note a couple of extra considerations that you don't need to worry about with local files. You need to know the connection string to use, which involves host, port, database name, username, and password. You also need permission to connect to the database and tables in question. If you're not managing your own database server, then you might need to talk to the database administrator to set all of this up. The following example connects to the geodata database being served by a PostgreSQL instance running on my local machine. It won't work for you unless you go to the trouble to install PostgreSQL and PostGIS, and then set up a database.

```
>>> pb.print_layers('PG:user=chris password=mypass dbname=geodata')
0: us.counties
1: global.countries
2: global.populated_places
3: time_zones
```

You see four layers here, but they're divided up into three different groups, or schemas. The time zones layer is in the default schema, counties is in the *us* schema, and the remaining two are in the *global* schema. Every user of the database could have

access to different schemas, and even different layers within a schema, depending on how the database administrator has set up the security.

As you can see, you can access PostGIS databases with OGR, but you can do many things with a PostGIS database that aren't covered in this book. If you're interested in learning more about it, take a look at *PostGIS in Action*, also published by Manning.

4.2.3 Folders as data sources (shapefiles and CSV)

In certain cases OGR will treat entire folders as data sources. Two examples of this are the shapefile and comma-delimited text file (.csv) drivers, which can be used to open either individual files or entire folders as data sources. If you use a folder, then each file inside of the folder is treated as a layer. If a folder contains a variety of file types, then the shapefile driver is used. For example, try listing the layers in the US folder:

```
>>> pb.print_layers(r'D:\osgeopy-data\US')
0: citiesx020 (Point)
1: cities_48 (Point)
2: countyp010 (Polygon)
3: roadtrl020 (LineString)
4: statep010 (Polygon)
5: states_48 (Polygon)
6: volcanx020 (Point)
```

Compare this list to the contents of the folder, and you'll see that it listed each of the shapefiles, but none of the others. The CSV driver is a little pickier, however, and wants all of the files in the folder to be CSV files. Although it won't work with the US folder, it works fine with the csv subfolder. Does this mean that you can't open a CSV file that's in a folder with a bunch of other files? Fortunately, no. All you have to do is treat the CSV file itself as a data source with only one layer. You can do the exact same thing with a shapefile by providing the name of the .shp file.

4.2.4 Esri file geodatabases

You Esri users out there might expect to see feature datasets inside file geodatabases treated like the schemas in PostGIS. If so, you'll be disappointed, because all you see are feature class names. Figure 4.3 shows what the natural_earth file geodatabase looks like in ArcCatalog, but the large_scale feature dataset name isn't included in the layer names that OGR uses.

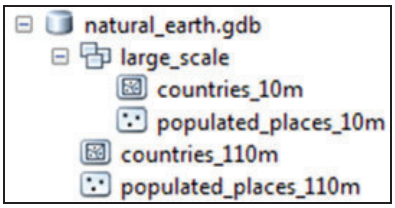


Figure 4.3 The natural_earth file geodatabase as seen in ArcCatalog

```
>>> pb.print_layers(r'D:\osgeopy-data\global\natural_earth.gdb')
0: countries_10m
1: populated_places_10m
2: countries_110m
3: populated_places_110m
```

Fortunately, you don't need the feature dataset name to access the layer, though; the feature class name works fine:

```
>>> ds = ogr.Open(r'D:\osgeopy-data\global\natural_earth.gdb')
>>> lyr = ds.GetLayer('countries_10m')
```

File geodatabases have two different drivers. You can read more about the differences on the OGR website, but one huge difference is that the read-only OpenFileGDB driver is compiled into OGR by default and the read/write FileGDB driver isn't because it requires a third-party library from Esri. If somebody gave you a file geodatabase that you needed to change but you didn't have access to the FileGDB driver, you could still use the OpenFileGDB driver to open the geodatabase and copy the data to a format that you could edit. This may not be ideal, but at least you have the option. For example, you could copy the countries_110m feature class in the natural earth geodatabase to a shapefile like this:

```
gdb_ds = ogr.Open(r'D:\osgeopy-data\global\natural_earth.gdb')
gdb_lyr = gdb_ds.GetLayerByName('countries_110m')
shp_ds = ogr.Open(r'D:\Temp', 1)
shp_ds.CopyLayer(gdb_lyr, 'countries_110m')
del shp_ds, gdb_ds
```

You haven't seen the `CopyLayer` method before. This allows you to easily copy the contents of an entire layer into a new data source or to the same data source but with a different layer name. To use it, you need to get the layer that you want to make a copy of and open the data source that you want to save the copy into. Then call `CopyLayer` on the data source that will get the copy, and pass it the original layer and a name for the new layer that will be created.

If you do have the Esri FileGDB driver, you can create new file geodatabases, and even feature datasets even though OGR doesn't show you feature dataset names. Listing 4.2 shows a function that imports all of the layers from a data source into a feature dataset within a file geodatabase, but note that this only works if you have the FileGDB driver. If you try to use this function without that driver installed, you'll get an error message that says AttributeError: 'NoneType' object has no attribute 'CreateDataSource'.

Listing 4.2 Function to import layers to a file geodatabase

```
def layers_to_feature_dataset(ds_name, gdb_fn, dataset_name):
    """Copy layers to a feature dataset in a file geodatabase."""
    in_ds = ogr.Open(ds_name)
    if in_ds is None:
        raise RuntimeError('Could not open datasource')
    gdb_driver = ogr.GetDriverByName('FileGDB')
    if os.path.exists(gdb_fn):
        gdb_ds = gdb_driver.Open(gdb_fn, 1)
    else:
        gdb_ds = gdb_driver.CreateDataSource(gdb_fn)
    if gdb_ds is None:
        raise RuntimeError('Could not open file geodatabase')
    options = ['FEATURE_DATASET=' + dataset_name]
    for i in range(in_ds.GetLayerCount()):
        lyr = in_ds.GetLayer(i)
        lyr_name = lyr.GetName()
        print('Copying ' + lyr_name + '...')
        gdb_ds.CopyLayer(lyr, lyr_name, options)
```

Open the geodatabase if it exists

Create the geodatabase if needed

Set the feature dataset name

Copy each layer

This function requires three parameters: the path to the original data source, the path to the file geodatabase, and the name of the feature dataset to copy the layers into. After opening the original data source, it checks to see if the file geodatabase exists. If it does, then the geodatabase is opened for writing. If it doesn't exist, it's created. Feature datasets are specified using layer-creation options, so then a list containing a single option for `FEATURE_DATASET` is created. After that, all of the layers in the original data source are looped over and copied into the geodatabase while keeping the same layer name (although they'll be renamed if naming conflicts arise in the geodatabase). If the `FEATURE_DATASET` layer-creation option wasn't provided, then the layer will be added to the file geodatabase, but it will be at the top level instead of in a feature dataset.

Now that you have this function, you could copy all of the shapefiles in a folder into a geodatabase like this:

```
layers_to_feature_dataset(
    r'D:\osgeopy-data\global', r'D:\Temp\osgeopy-data.gdb', 'global')
```

If you wanted to have the option of saving the feature classes to the top level of the geodatabase instead of in a feature dataset, you could modify this function so it doesn't pass the option list to `CopyLayer` if the `dataset_name` parameter is `None` or an empty string.

4.2.5 Web feature services

You can also access online services, such as *Web Feature Services* (WFS). Let's try this using a WFS hosted by the United States National Oceanic and Atmospheric Administration (NOAA) that serves out hazardous weather watches and advisories. Start with getting the list of available layers:

```
>>> url = 'WFS:http://gis.srh.noaa.gov/arcgis/services/watchWarn/' + \
...       'MapServer/WFSServer'
>>> pb.print_layers(url)
0: watchWarn:WatchesWarnings (MultiPolygon)
1: watchWarn:CurrentWarnings (MultiPolygon)
```

You can loop through these layers like the layers from other data sources, but all of the data are fetched immediately, so there could be quite a lag if the list has lots of features. It looks like the second layer only contains warnings, which are more severe than watches, so it should have less data. Let's find out what type of warning the first feature represents. I've discovered that things crash if I try to use `GetFeature` with an FID, but you can do it using `GetNextFeature`:

```
>>> ds = ogr.Open(url)
>>> lyr = ds.GetLayer(1)
>>> feat = lyr.GetNextFeature()
>>> print(feat.GetField('prod_type'))
Tornado Warning
```

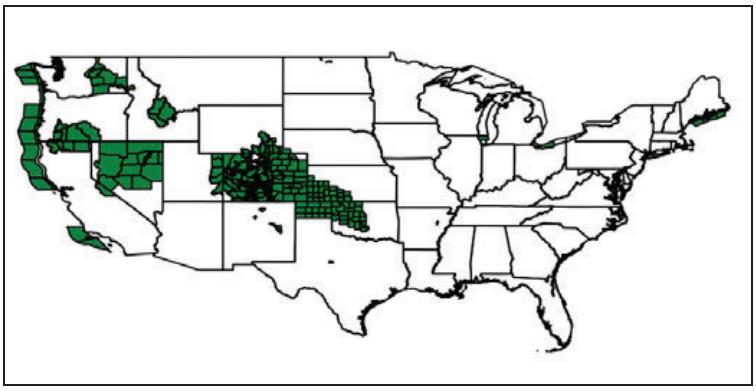


Figure 4.4
The WatchesWarnings layer from the NOAA web feature service. If you plot it, your results will differ because this layer shows real-time data.

I can recommend an easier and faster way to get only the first few features if that's all you want, however. Tack a `MAXFEATURES` parameter onto your URL, like this:

```
>>> url += '?MAXFEATURES=1'
>>> ds = ogr.Open(url)
>>> lyr = ds.GetLayer(1)
>>> lyr.GetFeatureCount()
1
```

You can also work with the geometries from a WFS. Figure 4.4 shows my results when I used VectorPlotter to draw the watchWarn:WatchesWarnings layer on top of states.

Let's do something a little different—save real-time data from a WFS and use it to build a simple web map using Folium, which is a Python module that creates Leaflet maps. If you have no idea what Leaflet is, that's okay, because you don't have to know anything about web mapping to work through this example. First you need to install Folium, though. On my Windows computer, I opened up a command prompt and used pip to install Folium and Jinja2 (another module that Folium requires in order to work) for Python 3.3 like this:

```
C:\Python33\Scripts\pip install Jinja2
C:\Python33\Scripts\pip install folium
```

If you're not familiar with installing Python modules via pip, please refer to the installation instructions in appendix A. Now let's look at the example script, which breaks things out into functions so code can be easily reused. Listing 4.3 contains a function to retrieve stream gauge data from a WFS and save it as GeoJSON; a function to make the web map showing these stream gauges; a function to get a geometry so that the map focuses on a single state instead of the whole country; and a couple of helper functions to format data for the WFS request and the map.

Listing 4.3 Create a web map from WFS data

```
import os
import urllib
from osgeo import ogr
import folium
```

```
def get_bbox(geom):
    """Return the bbox based on a geometry envelope."""
    return '{0},{2},{1},{3}'.format(*geom.GetEnvelope())

def get_center(geom):
    """Return the center point of a geometry."""
    centroid = geom.Centroid()
    return [centroid.GetY(), centroid.GetX()]

def get_state_geom(state_name):
    """Return the geometry for a state."""
    ds = ogr.Open(r'D:\osgeopy-data\US\states.geojson')
    if ds is None:
        raise RuntimeError(
            'Could not open the states dataset. Is the path correct?')
    lyr = ds.GetLayer()
    lyr.SetAttributeFilter('state = "{0}"'.format(state_name))
    feat = next(lyr)
    return feat.geometry().Clone()

def save_state_gauges(out_fn, bbox=None):
    """Save stream gauge data to a geojson file."""
    url = 'http://gis.srh.noaa.gov/arcgis/services/ahps_gauges/' + \
          'MapServer/WFSServer'
    parms = {
        'version': '1.1.0',
        'typeNames': 'ahps_gauges:Observed_River_Stages',
        'srsName': 'urn:ogc:def:crs:EPSG:6.9:4326',
    }
    if bbox:
        parms['bbox'] = bbox
    try:
        request = 'WFS:{0}?{1}'.format(url, urllib.urlencode(parms))
    except:
        request = 'WFS:{0}?{1}'.format(url, urllib.parse.urlencode(parms))
    wfs_ds = ogr.Open(request)
    if wfs_ds is None:
        raise RuntimeError('Could not open WFS.')
    wfs_lyr = wfs_ds.GetLayer(0)

    driver = ogr.GetDriverByName('GeoJSON')
    if os.path.exists(out_fn):
        driver.DeleteDataSource(out_fn)
    json_ds = driver.CreateDataSource(out_fn)
    json_ds.CopyLayer(wfs_lyr, '')

def make_map(state_name, json_fn, html_fn, **kwargs):
    """Make a folium map."""
    geom = get_state_geom(state_name)
    save_state_gauges(json_fn, get_bbox(geom))
    fmap = folium.Map(location=get_center(geom), **kwargs)
    fmap.geo_json(geo_path=json_fn)
    fmap.create_map(path=html_fn)

os.chdir(r'D:\Dropbox\Public\webmaps')
make_map('Oklahoma', 'ok.json', 'ok.html',
         zoom_start=7)
```

Get bounding box from geometry

Get center point from geometry

Get a state geometry

Save gauge WFS data to GeoJSON

Make the web map

Top-level code

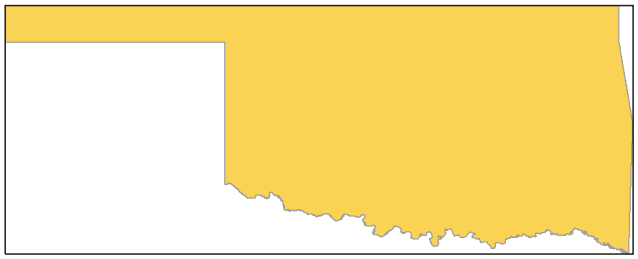


Figure 4.5 The line is the bounding box for the state of Oklahoma.

You can probably understand what the `get_state_geom` function does and how it does it, because you've seen the same process before. It takes a state name as a parameter, finds the corresponding feature in a layer, and returns the cloned geometry. The filename is hardcoded because you assume that the location of this state boundary file won't change.

The two helper functions are also simple. The `get_center` function takes a geometry, gets its centroid, and then returns the coordinates as a [y, x] list. The order might seem weird to you, but that's the order that Folium wants them in for the map.

The `get_bbox` function takes a geometry and returns its bounding coordinates as a string formatted like min_x,min_y,max_x,max_y. This is the format that a WFS uses to spatially subset results, and it's how you'll limit your gauge results to the bounding box of a state. This function takes advantage of the string formatting rules to rearrange the results of `GetEnvelope`, which returns a geometry's bounding box (figure 4.5) as a [min_x, max_x, min_y, max_y] list.

Now let's look at the slightly more complicated `save_state_gauges` function. Here you hardcode in the URL for a WFS that returns the observed river stages data from the Advanced Hydrologic Prediction Service. You also create a dictionary containing the parameters to be passed to the WFS. As you already know, the `typeNames` parameter is the name of the layer to retrieve data from. The `version` is the WFS version to use, and `srsName` specifies which coordinate system you'd like your data to be returned in. You can see the available options for this in the WFS's capabilities output, which you can get by tacking ?request=GetCapabilities onto the end of the service URL and visiting it in a web browser. For example, part of the output from http://gis.srh.noaa.gov/arcgis/services/ahps_gauges/MapServer/WFSServer?request=GetCapabilities looks like this:

```
<wfs:FeatureType>
    <wfs:Name>ahps_gauges:Observed_River_Stages</wfs:Name>
    <wfs:Title>Observed_River_Stages</wfs:Title>
    <wfs:DefaultSRS>urn:ogc:def:crs:EPSG:6.9:4269</wfs:DefaultSRS>
    <wfs:OtherSRS>urn:ogc:def:crs:EPSG:6.9:4326</wfs:OtherSRS>
    <snip>
</wfs:FeatureType>
```

From this you can see that the default spatial reference system (`DefaultSRS`) is EPSG 4269, which happens to be unprojected data using the NAD83 datum. If that doesn't make much sense, don't worry about it for now, because you'll learn all about it in

chapter 8. All you need to know now is that web-mapping libraries generally want coordinates that use WGS84, which corresponds to EPSG 4326. Fortunately, that's listed as an `OtherSRS` option in the capabilities output, so you insert it into your parameters dictionary:

```
parms = {
    'version': '1.1.0',
    'typeNames': 'ahps_gauges:Observed_River_Stages',
    'srsName': 'urn:ogc:def:crs:EPSG:6.9:4326',
}
if bbox:
    parms['bbox'] = bbox
```

If the user provided a `bbox` parameter to the function, you also insert that into your dictionary. If a `bbox` parameter is provided to the WFS, it returns features that fall in that box instead of returning all of them. Remember that your `get_bbox` function creates a string in the correct format for this based on a geometry's bounding box.

Creating this dictionary wasn't absolutely necessary, because you could have built your query string the same way you did in earlier examples, but I think that using a dictionary makes it easier to see what parameters are being passed. It's easy to create the query string from the dictionary by using the `urlencode` function, which formats everything for you. In Python 2, this function lives in the urllib module, but in Python 3 it lives in urllib.parse, which is why you have the next step in a try/except block. You try to create the query string using the Python 2 function, but if that fails because the script was run with Python 3, then you do it the Python 3 way instead:

```
try:
    request = 'WFS:{0}?{1}'.format(url, urllib.urlencode(parms))
except:
    request = 'WFS:{0}?{1}'.format(url, urllib.parse.urlencode(parms))
```

After creating your query string, you use it to open a connection to the WFS and get the layer. You want to save the output to a local file this time, though, so then you create an empty GeoJSON data source. Data sources have a `CopyLayer` function that copies an existing layer into the data source; this existing layer can be from another data source altogether. You use that function to copy the data from the WFS into your new GeoJSON file:

```
json_ds.CopyLayer(wfs_lyr, '')
```

The second parameter to `CopyLayer` is the name for the new layer, but GeoJSON layers don't have names, so you pass a blank string. You could pass a real layer name, but it wouldn't do much good. When your function returns after creating the layer, the data sources go out of scope, so the files get closed automatically, which is why you don't bother to close them inside the function.

The last function you write is called `make_map`. It wants a state name along with filenames for the output GeoJSON and HTML files. It can also take other named

arguments that get passed to Folium, which allows you to pass optional Folium parameters without having to worry about them in your `make_map` function:

```
def make_map(state_name, json_fn, html_fn, **kwargs):
    """Make a folium map."""
    geom = get_state_geom(state_name)
    save_state_gauges(json_fn, get_bbox(geom))
    fmap = folium.Map(location=get_center(geom), **kwargs)
    fmap.geo_json(geo_path=json_fn)
    fmap.create_map(path=html_fn)
```

The basic outline is shown in figure 4.6, but the first thing this function does is get the geometry for the state of interest. Then it gets the `bbox` for the geometry and passes that, along with the output GeoJSON filename, to the function that saves the WFS data to file. Then it creates a Folium map centered on the geometry, and also uses any named arguments that the user might have passed in. Remember that `**` explodes a dictionary into key/value pairs, so all of the arguments are treated as if they're an exploded dictionary called `kwargs`. You can read about the optional parameters at http://folium.readthedocs.org/en/latest/. This map uses OpenStreetMap tiles as the basemap by default, but that's one of the things you can change.

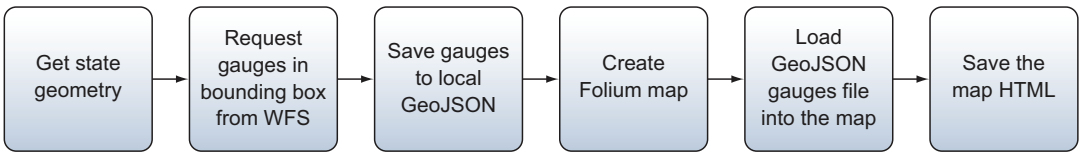


Figure 4.6 Tasks in the `make_map` function

After creating the basic map, the contents of the GeoJSON file are added and the map is saved to the HTML filename provided by the user. All that's left is to use it.

```
os.chdir(r'D:\Dropbox\Public\webmaps')
make_map('Oklahoma', 'ok.json', 'ok.html',
         zoom_start=7)
```

I used a Dropbox folder so that I could view the output on the web using the Dropbox public link functionality. You probably won't have much luck viewing the output straight from your local drive without using a web server. If you don't have something like Dropbox you can use, check out the sidebar to learn how to start up a simple Python web server on your local machine instead. I wanted to make a map of Oklahoma, and I also passed one of those optional parameters, `zoom_start`, through to Folium. By default, Folium maps start with a zoom level of 10, which is zoomed in too far to see the entire state. A start level of 7 works much better for this example.

Python SimpleHTTPServer

Python ships with a simple web server that you can use for testing things out, although you probably shouldn't use it for production websites. The easiest way to use it is to open up a terminal window or command prompt, change to the directory that contains the files you want to serve, and then invoke the server from the command line.

For Python 2:

```
D:\>cd dropbox\public\webmaps
D:\Dropbox\Public\webmaps>c:\python27\python -m SimpleHTTPServer
```

For Python 3:

```
D:\>cd dropbox\public\webmaps
D:\Dropbox\Public\webmaps>c:\python33\python -m http.server
```

This will start up a web server running on your local port 8000, so you can get to it in a web browser at http://localhost:8000/. If a file called index.html is in the folder you started the server from (d:\dropbox\public\webmaps, in this case), then that page will automatically be displayed. Otherwise, a list of files in the folder will display, and you can click on one to see it. The URL for the Oklahoma example would be http://localhost:8000/ok.html.

Once you've run the script, you can get the Dropbox public link for ok.html and view it in a web browser. If all went well, it will look something like figure 4.7.

The map in figure 4.7 shows the location of stream gauges, but other than that, it's not too useful. Smaller markers would be nice, and so would popups that provide the gauge reading if you click on the marker. Unfortunately, I don't believe there's a way to do this by adding a GeoJSON file to the map directly, but it's not hard to do manually. Let's add a function to make custom markers, along with a couple of helper

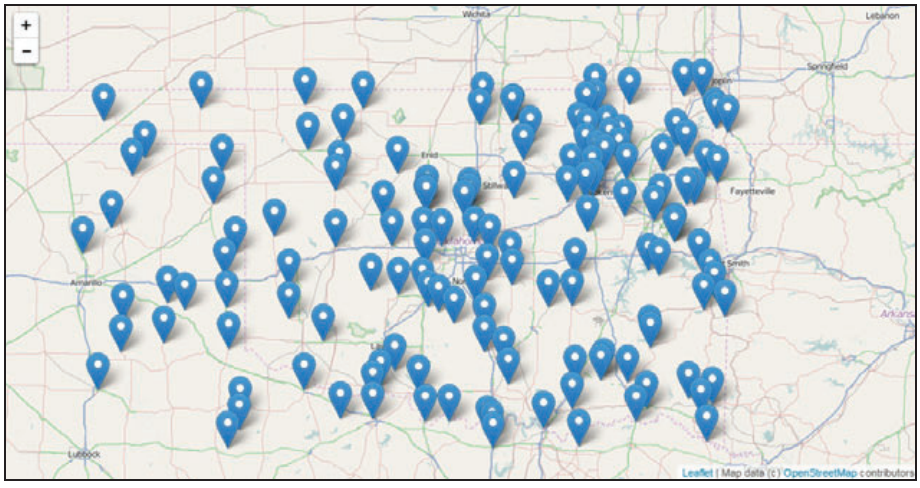


Figure 4.7 A simple Folium map made with a GeoJSON file

functions, and then change the `make_map` function to use those instead of adding the GeoJSON straight to the map.

Listing 4.4 Custom markers for a Folium map

```
colors = {
    'action': '#FFFF00',
    'low_threshold': '#734C00',
    'major': '#FF00C5',
    'minor': '#FFAA00',
    'moderate': '#FF0000',
    'no_flooding': '#55FF00',
    'not_defined': '#B2B2B2',
    'obs_not_current': '#B2B2B2',
    'out_of_service': '#4E4E4E'
}

def get_popup(attributes):
    """Return popup text for a feature."""
    template = '''{location}, {waterbody}</br>
                  {observed} {units}</br>
                  {status}'''
    return template.format(**attributes)

def add_markers(fmap, json_fn):
    ds = ogr.Open(json_fn)
    lyr = ds.GetLayer()
    for row in lyr:
        geom = row.geometry()
        color = colors[row.GetField('status')]
        fmap.circle_marker([geom.GetY(), geom.GetX()],
                           line_color=color,
                           fill_color=color,
                           radius=5000,
                           popup=get_popup(row.items()))

def make_map(state_name, json_fn, html_fn, **kwargs):
    """Make a folium map."""
    geom = get_state_geom(state_name)
    save_state_gauges(json_fn, get_bbox(geom))
    fmap = folium.Map(location=get_center(geom), **kwargs)
    add_markers(fmap, json_fn)
    fmap.create_map(path=html_fn)

os.chdir(r'D:\Dropbox\Public\webmaps')
make_map('Oklahoma', 'ok2.json', 'ok2.html',
         zoom_start=7, tiles='Stamen Toner')
```

Colors based on flood status

Create popup text for a feature

Add markers to the map

Use your new function

The first thing you do here is set up colors to use. These come from the online legend for this map service, which is available at http://gis.srh.noaa.gov/arcgis/rest/services/ahps_gauges/MapServer/0. The keys in the `colors` dictionary are possible values in the `Status` attribute field, and the values are hex strings that describe a color.

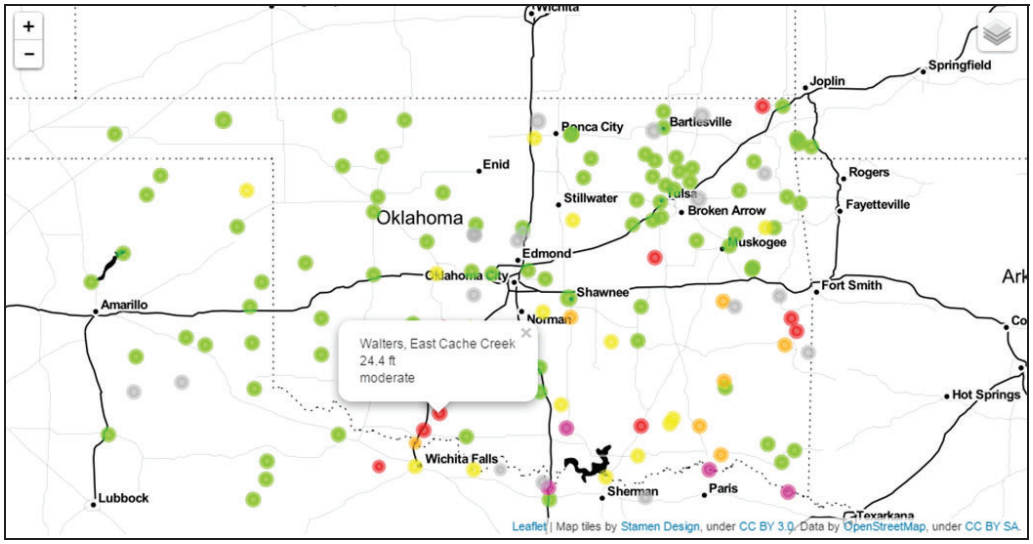


Figure 4.8 A nicer map created by manually constructing colored markers with popups

The `get_popup` function creates an HTML string by exploding the attributes dictionary for a feature and inserting the values in the corresponding placeholders in a template string. For example, the value from the `Location` field would get inserted in place of "{location}" in the template string.

The markers are created in the `add_markers` function, which loops through the GeoJSON layer and creates a marker for each point in the layer. This uses the Folium `circle_marker` function, which wants a [y, x] list as its first argument. This is where the marker will be placed on the map. You used a different color based on the flood status at that location, and also added a popup to go along with the marker. The `radius` parameter is the marker radius in pixels. Yours are a little larger than the default.

The last steps are to change the `make_map` function so that it calls `add_markers` instead of `geo_json`, and then to create a new map. This time you use Stamen Toner tiles instead of OpenStreetMap, mostly because the markers are easier to see that way. Your output should look like figure 4.8, and if you click on a marker, you'll see a popup containing the relevant information.

Although it isn't the subject of this book, I hope you enjoyed the short foray into web mapping. If you didn't know anything on the subject and are anything like me, you now have another item on your "to learn" list.

4.3 Testing format capabilities

As mentioned earlier, not all operations are available with all data formats and drivers. How do you find out what's allowed on your data, other than trying it and crossing your fingers that your code doesn't crash? Fortunately, drivers, data sources, and layers are all willing to convey that information if you ask. Table 4.1 shows which capabilities you can check for each of these data types.

Table 4.1 Constants used for testing capabilities

Driver capabilities	OGR constant
Create new data sources	`ODrCCreateDataSource`
Delete existing data sources	`ODrCDeleteDataSource`
`DataSource` capabilities	**OGR constant**
Create new layers	`ODsCCreateLayer`
Delete existing layers	`ODsCDeleteLayer`
Layer capabilities	**OGR constant**
Read random features using `GetFeature`	`OLCRandomRead`
Add new features	`OLCSequentialWrite`
Update existing features	`OLCRandomWrite`
Supports efficient spatial filtering	`OLCFastSpatialFilter`
Has an efficient implementation of `GetFeatureCount`	`OLCFastFeatureCount`
Has an efficient implementation of `GetExtent`	`OLCFastGetExtent`
Create new fields	`OLCCreateField`
Delete existing fields	`OLCDeleteField`
Reorder fields in the attribute table	`OLCReorderFields`
Alter properties of existing fields	`OLCAlterFieldDefn`
Supports transactions	`OLCTransactions`
Delete existing features	`OLCDeleteFeature`
Has an efficient implementation of `SetNextByIndex`	`OLCFastSetNextByIndex`
Values of string fields are guaranteed to be UTF-8 encoding	`OLCStringsAsUTF8`
Supports ignoring fields when fetching feature data, which can speed up data access	`OLCIgnoreFields`

To check for a given capability, all you have to do is call the `TestCapability` function on a driver, data source, or layer, and pass a constant from table 4.1 as a parameter. The function will return `True` if that operation is allowed and `False` if it isn't. Try using this to determine if you can add new shapefiles to a folder:

```
>>> dirname = r'D:\ osgeopy-data\global'
>>> ds = ogr.Open(dirname)
>>> ds.TestCapability(ogr.ODsCCreateLayer)
False
>>> ds = ogr.Open(dirname, 1)
>>> ds.TestCapability(ogr.ODsCCreateLayer)
True
```

Data source is opened read-only

Data source is opened for writing

As you probably could've guessed, you're allowed to create new layers when the folder has been opened for writing, but not when it has been opened read-only. How could you use this information to make sure you didn't attempt to do something that would cause an error? You can modify your code to add checks before you try to do any editing:

```
ds = ogr.Open(new_fn, 1)
if ds is None:
    sys.exit('Could not open {0}.'.format(new_fn))
lyr = ds.GetLayer(0)

if not lyr.TestCapability(ogr.OLCCreateField):
    raise RuntimeError('Cannot create fields.')

lyr.CreateField(ogr.FieldDefn('ID', ogr.OFTInteger))
```

Check that fields can be added

This snippet will raise an error and not continue if you aren't allowed to add fields to the layer. You could catch and handle this error if you needed to, or let it bail out. If you don't want to handle the errors, the biggest reason for checking beforehand is to make sure that all edits are possible before you start.

For example, what if a layer supported editing fields but not deleting features, and you wanted to do both? If you edited the fields before deleting the features, then part of your changes would take place (the field edits) before your code crashed when trying to delete features. Obviously, this is a problem if you want all or none when it comes to your edits. If partial edits don't bother you, then you may not want to worry about this issue, but you can avoid the problem by checking capabilities beforehand and not proceeding if you're not allowed to make all of your changes.

Another option, if partial edits are okay in your book but you still want to handle errors instead of letting the script crash, is to use OGR exceptions. You wouldn't need to add any code to test capabilities, but you'd need to remember to add `ogr.UseExceptions()` somewhere early in your script. Using this approach, the attempt to delete a feature would still fail, but it then throws a `RuntimeError` that you could catch.

A function in the ospybook module called `print_capabilities` will print what capabilities a driver, data source, or layer supports. Here's how to use it from the Python interactive window:

```
>>> driver = ogr.GetDriverByName('ESRI Shapefile')
>>> pb.print_capabilities(driver)
*** Driver Capabilities ***
ODrCCreateDataSource: True
ODrCDeleteDataSource: True
```

Because this function only prints out information, you can't use it in your code to determine what action to take based on available capabilities. You can use it in an interactive window to determine what actions were allowed on an object, though.

4.4 Summary

- The vector file format you choose to use might depend on the application. You might go with GeoJSON for making a web map, but use shapefiles or PostGIS for data analysis.
- Perhaps the most popular data transfer format is the shapefile because it's simple, the specifications are public, and it has been around for a long time.
- Formats based on databases, such as SpatiaLite, PostGIS, and Esri geodatabases, tend to be more efficient and support more features than other vector formats.
- Although the syntax for opening various data source types differs, once you have the data source open, you can access the layers and features the same way no matter the source.
- Multiple layers in a data source can be different from one another. For example, they can have different geometry types, attribute fields, spatial extents, and spatial reference systems.
- You can use `TestCapability` to determine which edits are allowed on your dataset.

5 Filtering data with OGR

This chapter covers

- Efficiently selecting features using attribute values
- Using spatial location to select features
- Joining attribute tables from different layers

Back in chapter 3, you learned how to iterate through all of the features in a layer and use attribute values for each one to determine if it was interesting. You've got easier ways to throw out features that you don't want, however, and that's where filters come in. With filters you can easily select features that match specific criteria, such as all animal GPS locations from a certain day or all crabapple trees from a city tree inventory. Filters also let you limit features by spatial extent, so you could limit your crabapple trees to a specific neighborhood, or GPS locations to those within a kilometer of an animal feeding station. Filtering your data like this makes it easy to extract or process only the features you're interested in. I've used these techniques to extract features such as city boundaries for a single county from a larger dataset, or to extract highways and freeways from road datasets, while ignoring the smaller residential roads.

You can also use SQL queries to join attribute tables together from different layers. For example, if you had a layer containing all of the locations of your store

franchises and each feature had an attribute denoting the city that the store was in, then you could join this layer with one containing cities. If the city layer contained demographic information for each city, then that data would be associated with the store data, and you could easily compare demographics between stores.

DEFINITION SQL is short for Structured Query Language, although you'll rarely see it written out like that. If you've used a relational database, then you've probably used SQL, even if you didn't realize it. For example, if you build a graphical query in Microsoft Access, it still builds a SQL query behind the scenes, and you can see it if you switch to SQL View. SQL is featured more prominently in other database software such as PostgreSQL.

5.1 Attribute filters

If you need to limit the features by values contained in one or more attribute fields, then you want an attribute filter. To set one of these filters, you need to come up with a conditional statement that's much like the `WHERE` clause in a SQL statement. You compare the value of an attribute field to another value, and then all features where that comparison is true are returned. The standard logical operators, such as `=`, `!=`, `<>`, `>`, `<`, `>=`, and `<=`, allow you to use statements such as the following:

```
'Population < 50000'
'Population >= 25000'
'Type_code != 7'
'Name = "Cairo"'
"Name = 'Moscow'"
'Name != "Tokyo"'
```

Numeric comparisons don't require quotes around the number

String comparisons require either single or double quotes

You can probably guess what these comparisons do; they all test for equality or inequality. Notice that if you're comparing strings, you need to put quotes around the string values, but they can be either single or double. Make sure they're different from the quotes you use to surround the entire query string, or else you'll end your string prematurely and get a syntax error. Don't use quotes with numbers, because that turns them into string values, and you won't get the comparison you were expecting. Another thing you might have noticed is that you use a single equal sign to test for equality, which isn't the way programming languages typically work. But that's the way SQL does things, so who are we to argue? In addition, if you want to test if something doesn't equal another value, you can use either `!=` or `<>`.

You can also combine statements using `AND` or `OR`:

```
'(Population > 25000) AND (Population < 50000)'
'(Population > 50000) OR (Place_type = "County Seat")'
```

The first of these selects features with a population value greater than 25,000 but less than 50,000. The second selects features that either have a population greater than 50,000 or are county seats (or both).

Conditions can be negated using `NOT`, and `NULL` is used to indicate a null or no data value in the attribute table:

```
'(Population < 50000) OR NOT (Place_type = "County Seat")'
'County NOT NULL'
```

That first example selects features that either have a population less than 50,000 or aren't county seats. Again, a feature will be selected if it meets one or both of those conditions. The second example selects features that have a value for the `County` attribute.

If you want to check if a value is between two other values, you can use `BETWEEN` instead of two different comparisons joined with `AND`. For example, the following two statements are equivalent, and both select features with a population between 25,000 and 50,000:

```
'Population BETWEEN 25000 AND 50000'
'(Population > 25000) AND (Population < 50000)'
```

You have an easy way to check if a value is equal to one of several different values. Once again, both of these select features where the `Type_code` value is 4, 3, or 7:

```
'Type_code IN (4, 3, 7)'
'(Type_code = 4) OR (Type_code = 3) OR (Type_code = 7)'
```

This also works for strings:

```
'Place_type IN ("Populated Place", "County Seat")'
```

Last, you can compare strings using the normal logical operators (a is less than c), or you can do fancier, case-insensitive, string matching using `LIKE`. This allows you to use wildcards to match any character in a string. An underscore matches any single character and a percent sign matches any number of characters. Table 5.1 shows examples, and this is how you'd use them:

```
'Name LIKE "%Seattle%"'
```

Table 5.1 Match examples using the LIKE operator

Pattern	Matches	Doesn't match
_eattle	Seattle	Seattle WA
Seattle%	Seattle, Seattle WA	North Seattle
%Seattle%	Seattle, Seattle WA, North Seattle	Tacoma
Sea%le	Seattle	Seattle WA
Sea_le	Seatle (note misspelling)	Seattle

If you want to read more about the SQL syntax available in OGR, check out the online documentation at http://www.gdal.org/ogr_sql.html and http://www.gdal.org/ogr_sql_sqlite.html. But for now, let's see how to put this newfound information to use. It will definitely be more fun if you fire up a Python interactive window for testing this out, because you can use the `VectorPlotter` class to interactively draw your

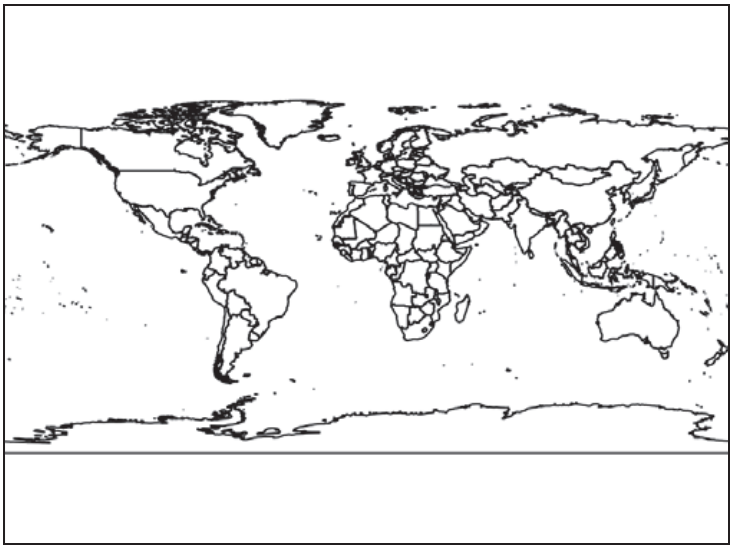


Figure 5.1
The ne_50m_admin_0_countries shapefile layer in the global data folder, with no filters applied

selections. After configuring an interactive vector plotter, open the global data folder and grab the low-resolution countries layer:

```
>>> ds = ogr.Open(r'D:\osgeopy-data\global')
>>> lyr = ds.GetLayer('ne_50m_admin_0_countries')
```

Then plot out the features, but be patient if it takes it a few seconds to draw the output shown in figure 5.1, since it has a fair amount of data to plot. Remember that setting `fill=False` tells it to draw only country outlines.

```
>>> vp.plot(lyr, fill=False)
```

Now inspect the layer attributes by printing out the names of the first few features:

```
>>> pb.print_attributes(lyr, 4, ['name'], geom=False)
FID    name
0      Aruba
1      Afghanistan
2      Angola
3      Anguilla
4 of 241 features
```

Notice that the feature IDs (FIDs) are in order and also the fact that there are 241 features in the layer. Now find out how many of those are in Asia by using an attribute filter. To do this, pass a conditional statement to `SetAttributeFilter`:

```
>>> lyr.SetAttributeFilter('continent = "Asia"')
0
>>> lyr.GetFeatureCount()
53
```

Now the layer thinks it has only 53 features. The zero that got spit out when you called `SetAttributeFilter` means that the query executed successfully. Now that you have

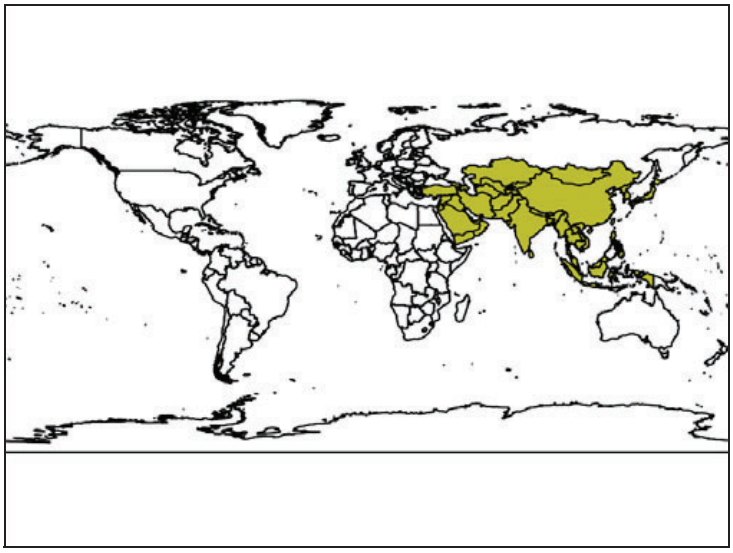


Figure 5.2 An attribute filter that selects countries in Asia has been applied to the countries layer shown in figure 5.1, and the results are plotted on top of the original.

selected the countries in Asia, try drawing them in yellow; your result should look like figure 5.2:

```
>>> vp.plot(lyr, 'y')
```

> **NOTE TO PRINT BOOK READERS: COLOR GRAPHICS** Many graphics in this book are best viewed in color. The eBook versions display the color graphics, so they should be referred to as you read. To get your free eBook in PDF, ePub, and Kindle formats, go to https://www.manning.com/books/geoprocessing-with-python to register your print book.

You can look a little more closely at what's happening with the filter by printing attributes for the first few features:

```
>>> pb.print_attributes(lyr, 4, ['name'], geom=False)
FID    name
1      Afghanistan
7      United Arab Emirates
9      Armenia
17     Azerbaijan
4 of 53 features
```

Huh. Now you're missing a bunch of FIDs. That's because those features aren't in Asia, so they're ignored while iterating through the layer. Getting features by specific FID doesn't honor the filter, however, because features aren't truly being deleted, and therefore the FID values don't change. You can prove it to yourself by getting a feature or two using FIDs:

```
>>> lyr.GetFeature(2).GetField('name')
'Angola'
```

You can see from this that even though Angola doesn't show up when you iterate through the filtered layer, it's still there. It should be obvious to you now that looping

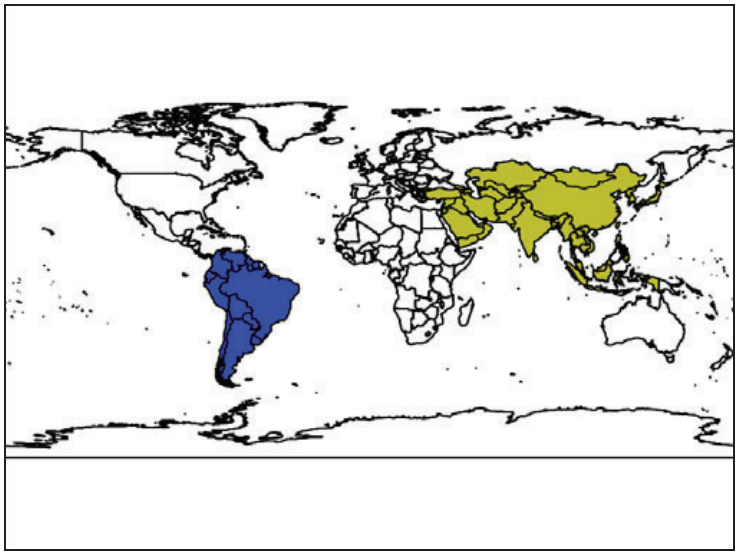


Figure 5.3 An attribute filter that selects countries in South America has been applied to the countries layer and the results plotted on top of the previous data from figure 5.2.

through a filtered layer using specific FIDs is a bad idea and you won't get the desired results. Instead, you need to iterate through the layer using a `for` loop.

If you set another attribute filter, it doesn't create a subset of the currently filtered features. Instead, the new filter is applied to the entire layer. To illustrate this, try applying a new filter that selects the countries in South America, and then draw them in blue, which results in the shading you see in figure 5.3.

```
>>> lyr.SetAttributeFilter('continent = "South America"')
>>> vp.plot(lyr, 'b')
```

You can, however, use both attribute and spatial filters together to refine your results, and you'll see an example of that in the next section. To clear out the attribute filter and get all 241 features back, simply pass `None` to `SetAttributeFilter`:

```
>>> lyr.SetAttributeFilter(None)
>>> lyr.GetFeatureCount()
241
```

Removing the filter also resets the current feature back to the beginning, as if you had just opened the layer.

5.2 Spatial filters

Spatial filters let you limit the features by spatial extent rather than attribute value. These filters can be used to select features within another geometry or inside a bounding box. For example, if you had a dataset of global cities with no attribute indicating the country that the cities are in, but you had another dataset with the same spatial reference system that contained the boundary of Germany, you could use a spatial filter to select the German cities.

Spatial reference systems and spatial filters

The geometries or coordinates used for spatial filtering must use the same spatial reference system as the layer you're trying to filter. Why is this? Pretend for a moment that you have a layer that uses a Universal Transverse Mercator (UTM) spatial reference system. Coordinates in that layer would be large numbers, much different than the latitude and longitude values we're all familiar with. This means that they wouldn't align if plotted on top of each other, and they'd appear to have non-overlapping spatial extents. For example, the UTM easting and northing coordinates for the capitol building in Salt Lake City, UT, are approximately 425045 and 4514422, but the corresponding longitude and latitude are -111.888 and 40.777. Those coordinates are awfully different from each other, and they wouldn't overlay on each other unless one of them was transformed to the same spatial reference system as the other.

Try selecting cities in Germany using the natural earth shapefiles. After setting up a vector plotter in an interactive window, open the folder data source and get the countries layer. Then use an attribute filter to limit the countries to Germany and grab the corresponding feature and geometry:

```
>>> ds = ogr.Open(r'D:\osgeopy-data\global')
>>> country_lyr = ds.GetLayer('ne_50m_admin_0_countries')
>>> vp.plot(country_lyr, fill=False)
>>> country_lyr.SetAttributeFilter('name = "Germany"')
>>> feat = country_lyr.GetNextFeature()
>>> germany = feat.geometry().Clone()
```

You can assume, in this case, that the attribute filter will return one and only one feature, so using `GetNextFeature` will get the first and only feature in the filtered results. Then you grab the geometry and clone it so that you can use the geometry even after

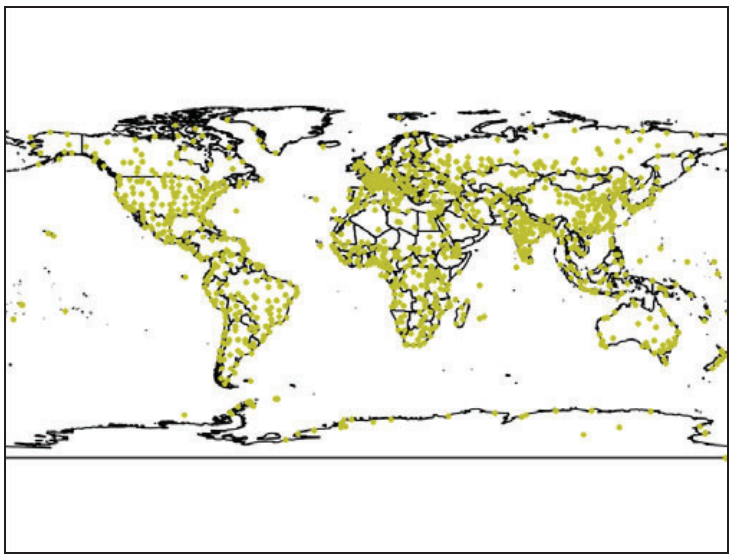


Figure 5.4 All of the cities in the populated_places layer in the natural earth dataset

the feature is removed from memory. Oh, and you also plot the world countries before applying the filter so that you have context for the cities later on. Now open the populated places layer and plot all cities (see figure 5.4) as yellow dots:

```
>>> city_lyr = ds.GetLayer('ne_50m_populated_places')
>>> city_lyr.GetFeatureCount()
1249
>>> vp.plot(city_lyr, 'y.')
```

The call to `GetFeatureCount` indicates there are 1,249 city features in the full layer. Now try applying a spatial filter by passing the `germany` geometry that you got earlier to `SetSpatialFilter`, and then plot the resulting cities as large dots:

```
>>> city_lyr.SetSpatialFilter(germany)
>>> city_lyr.GetFeatureCount()
5
>>> vp.plot(city_lyr, 'bo')
```

Now the layer claims to have only five features, so five cities fall within the German boundary polygon. You can also see from your plot that the circles fall in the correct geographical area. You can use the Zoom to rectangle tool on the bottom of the plot window to zoom in on Germany if you'd like (figure 5.5).

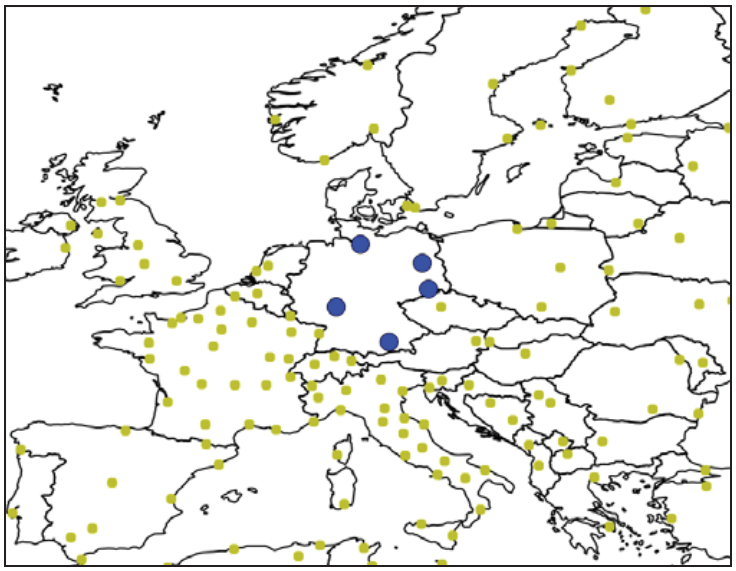


Figure 5.5 A spatial filter has been applied to the populated_places layer to limit the features to those within the boundaries of Germany. These filtered points are shown in as large dots.

To clone or not to clone?

Geometry objects have a `Clone` function, which makes a copy of the object. Why would you want to use this? When you get a geometry from a feature, that geometry is still associated with that feature. If that feature is then deleted (or the variable is populated

(continued)

with a different feature), then the geometry is no longer useable. In fact, if you try to use it, Python will crash instead of spit out an error. This problem is easy to solve, however, by cloning the geometry. Now you can store a copy of the feature or geometry that's no longer associated with other objects and will live on even if the parent objects disappear. Want to see this in action? Try this in an interactive window:

```
>>> ds = ogr.Open(r'D:\osgeopy-data\global\natural_earth_50m.sqlite')
>>> lyr = ds.GetLayer('countries')
>>> feat = lyr.GetNextFeature()
>>> geom = feat.geometry()
>>> geom_clone = feat.geometry().Clone()
>>> feat = lyr.GetNextFeature()
>>> print(geom_clone.GetArea())
0.014118879217099978
>>> print(geom.GetArea())
```

Delete the feature that the geometries came from

Python crashes

In this example, the `geom` variable holds a `Geometry` object that's still owned by the `Feature` object stored in the `feat` variable, but the `geom_clone` variable holds a geometry that has been disassociated from that feature. After you populate the `feat` variable with a different feature, you can still use the `geom_clone` geometry, but not the object stored in the `geom` variable, because you no longer have a handle to the feature that it came from.

Incidentally, this is related to why all of these examples would also cause Python to crash:

```
feat = ogr.Open(fn, 0).GetLayer(0).GetNextFeature()
# or
lyr = ogr.Open(fn, 0).GetLayer(0)
feat = lyr.GetNextFeature()
# or
ds = ogr.Open(fn, 0)
lyr = ds.GetLayer(0)
del ds
feat = lyr.GetNextFeature()
```

In each case, the data source has gone out of scope or been deleted before you try to use the layer. But the layer is associated with the data source and becomes unusable once the data source is gone, the same way a geometry becomes unusable if its parent feature disappears. You should never close your data source if you still need access to the layer.

As promised, you now get to combine a spatial and an attribute query. Further refine your selection by finding the cities with a population over 1,000,000, and draw them as the squares shown in figure 5.6:

```
>>> city_lyr.SetAttributeFilter('pop_min > 1000000')
>>> city_lyr.GetFeatureCount()
3
>>> vp.plot(city_lyr, 'rs')
```

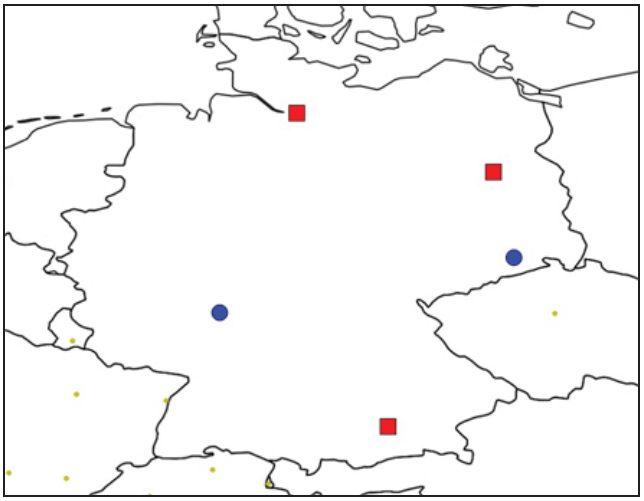


Figure 5.6 An attribute filter has been combined with a spatial filter to select the German cities with a population greater than 1,000,000 people. The selected features are shown as squares instead of circles.

Judging from these results, there are three German cities with populations of more than 1,000,000 people. Figure 5.6 shows the output plot zoomed in on Germany so you can see these features. But what if you decide that you want to know how many cities exist in the entire world with a population that large? All you have to do is remove the spatial filter by passing `None` to `SetSpatialFilter`. Note that the attribute filter will still be in effect. Go ahead and try it, drawing the results as triangles:

```
>>> city_lyr.SetSpatialFilter(None)
>>> city_lyr.GetFeatureCount()
246
>>> vp.plot(city_lyr, 'm^', markersize=8)
```

And now you know where the largest cities in the world are (figure 5.7).

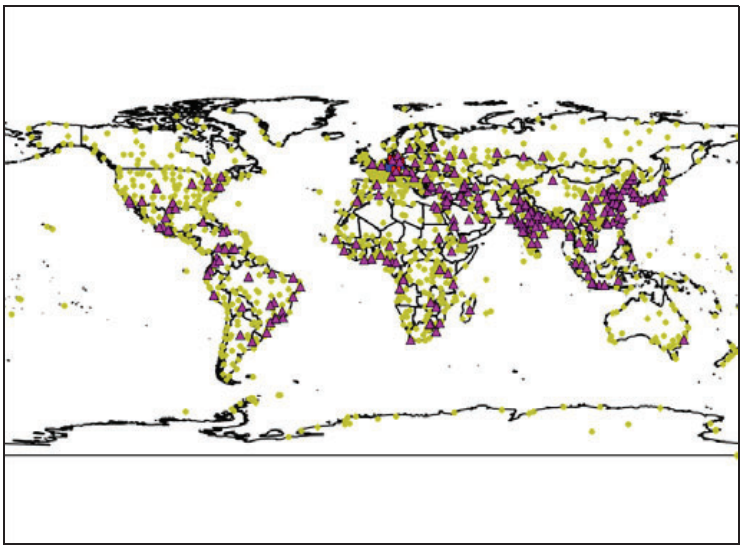


Figure 5.7 The spatial filter has been removed, but the attribute filter is still in effect, so now all of the cities in the world with a population of more than 1,000,000 are drawn as triangles over the top of the original dots for all cities.

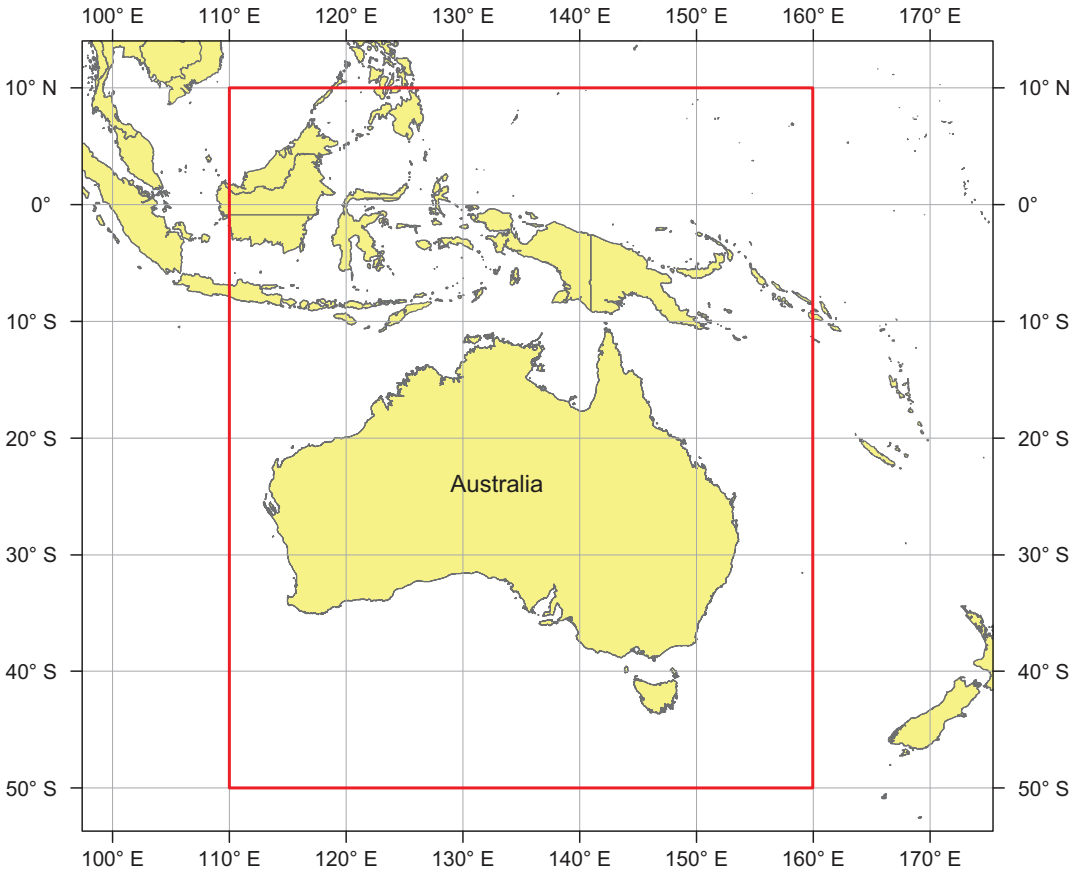


Figure 5.8 The minimum and maximum x and y values for the rectangle surrounding Australia can be used to set a spatial extent on the global countries layer.

You're not completely out of luck if you'd like to filter features spatially but don't have a geometry to use. You can also use a rectangular extent by providing the minimum and maximum x and y coordinates:

```
SetSpatialFilterRect(minx, miny, maxx, maxy)
```

You can use this to select the countries that fall within the box shown in figure 5.8. Again, start by plotting all of the countries:

```
>>> vp.clear()
>>> country_lyr.SetAttributeFilter(None)
>>> vp.plot(country_lyr, fill=False)
```

Now plug in the bounding coordinates shown in figure 5.8:

```
>>> country_lyr.SetSpatialFilterRect(110, -50, 160, 10)
>>> vp.plot(country_lyr, 'y')
```

Now you should have a plot that looks similar to figure 5.9, with Australia and a few surrounding countries shaded in.

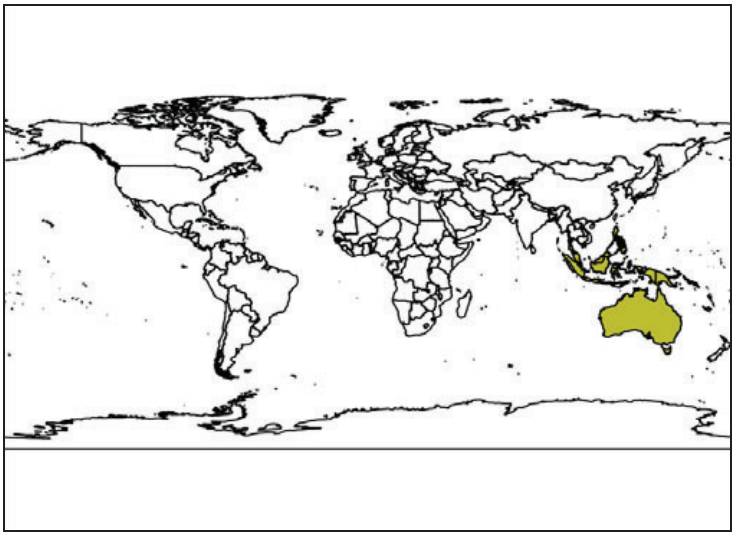


Figure 5.9 The shaded countries were selected using the rectangular extent shown in figure 5.8.

TIP To clear a spatial filter, whether it was created with a geometry or a bounding box, pass `None` to `SetSpatialFilter`. You can't clear the filter using `SetSpatialFilterRect`.

5.3 Using SQL to create temporary layers

If you're familiar with SQL, or are willing to learn, you can create more-complicated queries and do fun stuff using the `ExecuteSQL` function on a data source. This function applies to a data source instead of a layer because it allows you to use multiple layers if desired. It requires a SQL query and can optionally use a geometry as a spatial filter. In addition, you can also specify a different SQL dialect, but more on that later. Here's the signature:

```
ExecuteSQL(statement, [spatialFilter], [dialect])
```

- `statement` is the SQL statement to use.
- `spatialFilter` is an optional geometry to use as a spatial filter on the results. The default is no filter.
- `dialect` is a string specifying the SQL dialect to use. Available options are `OGRSQL` and `SQLite`. The default is to use the OGR dialect unless the data source has its own SQL engine (such as a SpatiaLite database).

This function is different from the filtering functions in that it returns a new layer containing the result set rather than only filtering features out of the existing layer. Let's look at a few examples using this technique, starting with a simple one that returns global countries sorted by population in descending order:

```
>>> ds = ogr.Open(r'D:\osgeopy-data\global')
>>> sql = '''SELECT ogr_geom_area as area, name, pop_est
...          FROM 'ne_50m_admin_0_countries' ORDER BY POP_EST DESC'''
>>> lyr = ds.ExecuteSQL(sql)
```

```
>>> pb.print_attributes(lyr, 3)
FID     Geometry        area                    name             pop_est
41      MULTIPOLYGON    950.9810937547769       China            1338612970.0
98      MULTIPOLYGON    278.3474038553223       India            1166079220.0
226     MULTIPOLYGON    1115.1781907153158      United States    313973000.0
3 of 241 features
```

As you can see from these results, the three most populous countries in the world are China, India, and the United States, in that order. The query returns each country's name and population attributes because you request them in the SQL statement. You also use the special `ogr_geom_area` field to get the area of each geometry (table 5.2), and the FID and geometry itself are returned automatically. This example uses the default OGR SQL dialect because shapefiles don't have any built-in SQL support.

Table 5.2 Special fields used in the OGR SQL dialect

Field	Returns
`FID`	The feature ID.
`OGR_GEOMETRY`	An OGR geometry type constant (see table 3.1). This is especially useful for data formats that support multiple geometry types in one layer.
`OGR_GEOM_WKT`	The well-known text (WKT) representation of the feature's geometry.
`OGR_GEOM_AREA`	The area of the feature's geometry. Returns zero for geometries with no area (for example, points or lines).
`OGR_STYLE`	The style string for the feature, if it exists. Very few applications use this.

If you're querying a data source that has its own SQL support, that native SQL version will be used. For example, if you have the SQLite driver, you could get the same information from the natural_earth_50m.sqlite database using the SQLite version of SQL. This dialect also allows you to limit the number of returned features, so you could limit the result set to the three countries with the highest populations:

```
>>> ds = ogr.Open(r'D:\osgeopy-data\global\natural_earth_50m.sqlite')
>>> sql = '''SELECT geometry, area(geometry) AS area, name, pop_est
...          FROM countries ORDER BY pop_est DESC LIMIT 3'''
>>> lyr = ds.ExecuteSQL(sql)
>>> pb.print_attributes(lyr)
FID     Geometry        area                    name             pop_est
0       MULTIPOLYGON    950.9810937547769       China            1338612970.0
1       MULTIPOLYGON    278.3474038553223       India            1166079220.0
2       MULTIPOLYGON    1115.1781907153158      United States    313973000.0
3 of 3 features
```

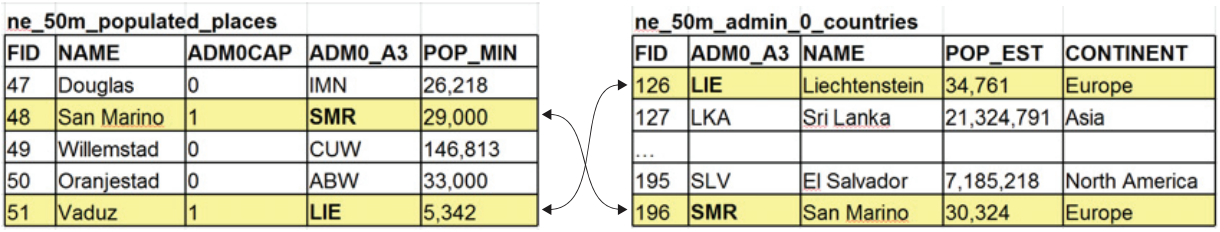
This time you could print attributes for the entire layer, because only three features are returned. You should also notice that now you use the `area` function instead of a special field name, and if you don't rename it with the `AS area` syntax, then it would be called `area(geometry)` instead. You also have to specifically request the geometry because the SpatiaLite engine doesn't return the geometry by default.

You can also use `ExecuteSQL` to join attributes from multiple layers. Take a look at this code and see if you can figure out what it's doing:

```
ds = ogr.Open(r'D:\osgeopy-data\global')
sql = '''SELECT pp.name AS city, pp.pop_min AS city_pop,
             c.name AS country, c.pop_est AS country_pop
         FROM ne_50m_populated_places pp
         LEFT JOIN ne_50m_admin_0_countries c
         ON pp.adm0_a3 = c.adm0_a3
         WHERE pp.adm0cap = 1'''
lyr = ds.ExecuteSQL(sql)
```

Rename the layers

The first thing to notice is that you use the ne_50m_populated_places and ne_50m_admin_0_countries shapefiles and rename them to `pp` and `c`, respectively. You do this by putting the alias directly after the layer name. This isn't necessary, of course, but does make your SQL a lot shorter because those layer names are pretty long. You also link these two layers together by using a join, which allows you to link tables using a shared attribute. Here you use a LEFT JOIN to keep all records in the table on the left (populated places), and if a matching record exists in the table on the right (countries), then you'll also get data from that record. But how does it figure out what matches? That's where the `ON` clause comes in. For each feature in `pp`, it takes the `adm0_a3` attribute value and tries to find a feature in the countries layer that has the same value for its `adm0_a3` field. See figure 5.10 for an illustration.



ne_50m_populated_places

FID	NAME	ADM0CAP	ADM0_A3	POP_MIN
47	Douglas	0	IMN	26,218
48	San Marino	1	SMR	29,000
49	Willemstad	0	CUW	146,813
50	Oranjestad	0	ABW	33,000
51	Vaduz	1	LIE	5,342

ne_50m_admin_0_countries

FID	ADM0_A3	NAME	POP_EST	CONTINENT
126	LIE	Liechtenstein	34,761	Europe
127	LKA	Sri Lanka	21,324,791	Asia
...				
195	SLV	El Salvador	7,185,218	North America
196	SMR	San Marino	30,324	Europe

Figure 5.10 An illustration of a SQL query that selects records from the populated places table where adm0cap equals 1, and then gets related data from the countries table based on the adm0_a3 field in both tables.

Now that you know what tables the data are coming from, go back to the beginning of the SQL statement and look at what attribute fields are being requested. You ask for the `NAME` and `POP_MIN` fields from the populated places layer, as well as the `NAME` and `POP_EST` fields from the countries layer. Because the fields from the two layers have the same names, it makes sense to rename them so that you can tell what's what. Last, you use a `WHERE` clause to limit the results to features that represent capital cities (adm0cap = 1).

This technique is handy if you want to see related data from multiple layers at the same time. Without this, you could query city populations and country populations

separately, but now you can see the country's population right beside the city's. To see this, look at the layer returned by this query:

```
pb.print_attributes(lyr, 3, geom=False)
FID     city            city_pop    country          country_pop
7       Vatican City    832         Vatican          832.0
48      San Marino      29000       San Marino       30324.0
51      Vaduz           5342        Liechtenstein    34761.0
3 of 200 features
```

I didn't print the geometry column because it wouldn't fit comfortably on the page, but because this uses the OGR SQL dialect, the geometry is returned automatically. But which one: the city or the country? It's the city, because that's the main table being used in the join, and corresponding country information is returned only if it existed for a city. You could plot the layer to prove it to yourself if you'd like.

Now check out a similar example using the SQLite dialect, but still shapefile data sources (you could use a SQLite database, of course, but I want to prove that the SQLite dialect will work with other data source types). See if you can spot the differences:

```
ds = ogr.Open(r'D:\osgeopy-data\global')
sql = '''SELECT pp.name AS city, pp.pop_min AS city_pop,
             c.name AS country, c.pop_est AS country_pop
         FROM ne_50m_populated_places pp
         LEFT JOIN ne_50m_admin_0_countries c
         ON pp.adm0_a3 = c.adm0_a3
         WHERE pp.adm0cap = 1 AND c.continent = "South America"'''
lyr = ds.ExecuteSQL(sql, dialect='SQLite')
pb.print_attributes(lyr, 3)
```

The most obvious difference is the inclusion of the `dialect` parameter to the `ExecuteSQL` function. But you also add one thing to the SQL that doesn't work with the OGR dialect. This time the results are limited to cities in South America by checking the value of the continent field in the countries layer. The OGR dialect doesn't support using fields from the joined table in the `WHERE` clause, so the only attributes allowed would be ones from the populated places layer. Also, because you need to specifically request geometries if you want them when using the SQLite dialect, no geometries are returned by this particular query. You could add them in by specifying `pp.geometry` along with the other fields.

If your version of OGR was built with SpatiaLite support (not only SQLite), you can also manipulate geometries within your SQL. Be warned that this could take a while, depending on what you try to do. As an example, if you have SpatiaLite support, try merging all of the counties in California into one big geometry. Start with drawing the individual counties so you have something to compare your results with:

```
>>> ds = ogr.Open(r'D:\osgeopy-data\US')
>>> sql = 'SELECT * FROM countyp010 WHERE state = "CA"'
>>> lyr = ds.ExecuteSQL(sql)
>>> vp.plot(lyr, fill=False)
```

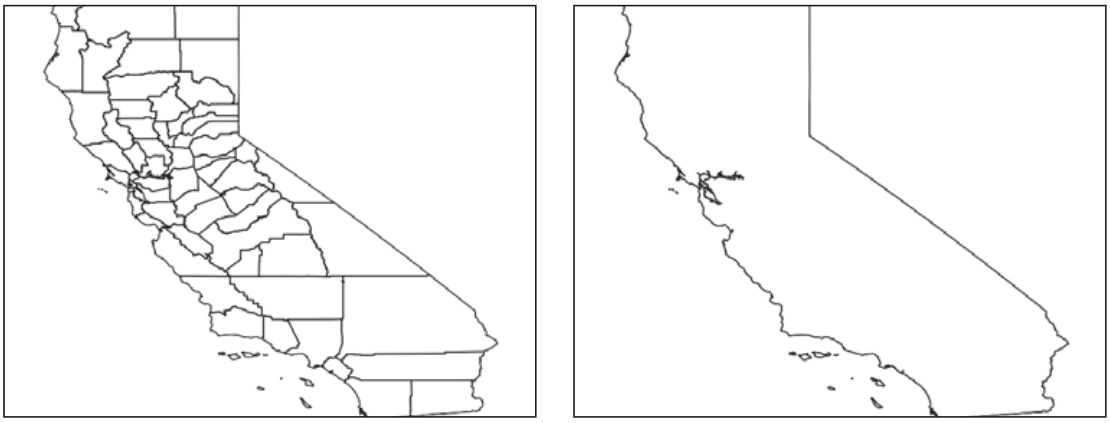


Figure 5.11 Part A, on the left, shows the counties in California drawn individually. Part B, on the other hand, shows the result of running the SpatiaLite `st_union` function on the counties. They're all joined together into one geometry.

This will draw a map of the counties in California, as shown in figure 5.11A. Now try using the SpatiaLite `st_union` function to merge all of the county polygons into one, as shown in figure 5.11B:

```
>>> sql = 'SELECT st_union(geometry) FROM countyp010 WHERE state = "CA"'
>>> lyr = ds.ExecuteSQL(sql, dialect='SQLite')
>>> vp.plot(lyr, 'w')
```

Geometry operations also work with data sources that have their own native SQL flavor and the ability to perform geometry manipulations. SpatiaLite and PostGIS are two obvious examples of this. For example, this is how you'd do the same thing with a PostGIS data source:

```
conn_str = 'PG:host=localhost user=chrisg password=mypass dbname=geodata'
ds = ogr.Open(conn_str)
sql = "SELECT st_union(geom) FROM us.counties WHERE state = 'CA'"
lyr = ds.ExecuteSQL(sql)
vp.plot(lyr)
```

Don't worry if you want to perform operations like this but aren't using PostGIS or SpatiaLite, because you'll learn how to do it without databases in the next chapter.

5.4 Taking advantage of filters

Remember back in chapter 3 when you copied all of the capital cities in a global shapefile into a new shapefile? You looped through each feature in the shapefile, checked the appropriate attribute, and copied the feature if it was a capital city. This whole process can be made much easier if the features you want can be selected with filters. Do you remember the `CopyLayer` method that was introduced in section 4.2.4? As a reminder, it copies an existing layer into a new data source. How do you think you

could use this to do something similar to the code back in listing 3.3, but much easier? Think about this problem for a minute and then look at the next example:

```
ds = ogr.Open(r'D:\osgeopy-data\global', 1)
in_lyr = ds.GetLayer('ne_50m_populated_places')
in_lyr.SetAttributeFilter("FEATURECLA = 'Admin-0 capital'")
out_lyr = ds.CopyLayer(in_lyr, 'capital_cities2')
```

Here the call to `CopyLayer` makes a copy of `in_lyr` in the `ds` data source. In this case, it happens to be the same data source as the original layer, but it could be any data source. Because you've already set an attribute filter on `in_lyr`, only the filtered features are copied. That's certainly easier than checking each one.

If you only want certain attributes, you could use a layer created using `ExecuteSQL`. Write a SQL query that pulls out the attributes you want and copy the results to a new layer:

```
sql = """SELECT NAME, ADM0NAME FROM ne_50m_populated_places
         WHERE FEATURECLA = 'Admin-0 capital'"""
in_lyr2 = ds.ExecuteSQL(sql)
out_lyr2 = ds.CopyLayer(in_lyr2, 'capital_cities3')
```

It should be obvious by now that you can simplify your life by taking advantage of filters and the `ExecuteSQL` function whenever possible.

5.5 Summary

- Attribute filters can be used to efficiently select specific features based on their attribute values.
- Spatial filters allow you to select features based on their location by using a bounding polygon or coordinates for a bounding box. The coordinates to set a spatial filter must use the same spatial reference system as the data to be filtered.
- Spatial and attribute filters can be combined.
- You can use SQL queries to create temporary layers made up of multiple layers joined on attribute values.
- You can't use objects once their owners go out of scope, so if you want to use a geometry after you've lost the handle to its feature, make sure you clone the geometry. Always keep your data source open if you want access to the layers. If you break one of these rules, Python will crash and burn.