1 Introduction

This chapter covers

- Introducing basic types of spatial data
- What is geoprocessing?
- Using QGIS

Humans have been making maps for far longer than we've been writing, and even the famed Lascaux caves in France have a star map on their walls. We know that ancient peoples all over the world used maps, including the Babylonians, Greeks, and Chinese. The art of cartography has evolved over the millennia, from cave walls as mediums to clay tablets, parchment, paper, and now digital. Maps have also gotten much more detailed, as well as accurate, as technology has been developed and improved. In fact, most of us would probably have a hard time recognizing the most primitive maps as maps at all.

It took mankind a long time to go from cave walls to mass-produced road maps, but the degree of change in the last few decades has been staggering. *Geographic Information Systems* (GISs) became more common and easier to use, giving more people the ability to both analyze spatial data and produce their own high-quality maps. Then came web mapping and services that allow users to make custom maps online and share them with the world. Many of us even carry devices in our pockets

that can display a map showing our current location and tell us how to get to a new restaurant that we want to try. Not only that, but the available data has also changed dramatically. Makers of those early maps would be blown away by our roadmaps overlaid on top of aerial photography and our talking GPS units.

Thanks to these recent advances in technology, along with free and open source tools, you have access to powerful software to work with your own data. This book aims to teach you the basic concepts of working with spatial data and how to do so with the Python programming language and a few open source tools. After reading this book, you'll write Python scripts to solve basic data analysis problems and have the background knowledge to answer more-complicated questions.

1.1 Why use Python and open source?

Several compelling reasons exist for using Python and open source tools for processing spatial data. First, Python is a powerful programming language that has the advantage of being much easier to learn than some other languages, and it's also easy to read. It's a good language to start with if you've never programmed before, and if you're coming from other languages, you'll probably find Python easy to pick up.

Learning Python is a good move, even if you never again use it for spatial analysis after reading this book. Many different Python modules are available for a wide range of applications, including web development, scientific data analysis, and 3D animation. In fact, geospatial applications are only a small subset of what Python is used for.

In addition, Python is multiplatform, so unless you've used an extra module that's specific to one operating system, a Python script that you write on one machine will run on any other machine, provided the required modules are installed. You can use your Linux box to develop a set of scripts and then give them to a colleague who uses Windows, and everything should work fine. You do need to install a Python interpreter to run the code, but those are freely available for major desktop operating systems.

Python ships with the core language and numerous modules that you can optionally use in your code. In addition, many more modules are available from other sources. For example, the *Python Package Index* (PyPI), available at https://pypi.python.org/pypi, lists more than 60,000 additional modules, all used for different purposes, and all free. That's not to say that everything Python is free, however. Several of you coming from a GIS background are no doubt familiar with *ArcPy*, which is a Python module that comes with *ArcGIS*, and is not useable without an ArcGIS license.

Not only is there an abundance of free Python packages, but many of them are also open source. Although many people associate open source software with software that doesn't cost money, that's only part of it. The real meaning is that the source code is made available for you to use if you wish. The fact that you have access to the source code means that nothing is a "black box" (if you want to take the time to learn what's inside the box), but also that you can modify the code to suit your needs. This is extremely liberating. I've used open source tools that didn't quite do what I wanted, so I tweaked the source code, recompiled, and then had a utility that did exactly what

I needed. This is impossible with proprietary software. These two types of freedom associated with open source software make it an attractive model.

Several different types of open source licenses exist, some of which not only allow you to modify the code as needed, but even allow you to turn around and sell your derived work without providing the source code and your modifications. Other licenses require that if you use the software, then your software must also be open source.

We'll cover a few popular open source Python modules for geospatial data in this book. Several were originally developed in other languages, but became so common and well respected that they were either ported to other languages, or bindings were developed so that they could be used in other languages. For example, the *Geospatial Data Abstraction Library* (GDAL) is an extremely popular C/C++ library for reading and writing spatial data, and bindings have been developed for Python, .NET, Ruby, and other languages. The GDAL library is even used by many proprietary software packages. Because of the library's widespread use, this book concentrates on GDAL/OGR. If you can learn to use this, then moving to other libraries shouldn't be difficult. In fact, several nice libraries are built on top of GDAL/OGR that are probably easier to use, but don't necessarily provide all of the functionality that's present in GDAL. See appendix A for installation instructions for the modules used in this book.

Another advantage to going with open source tools is that active user communities exist for some of these packages, and you may find that bugs and other issues are addressed much more quickly than with many proprietary software packages. You can even discuss the finer points of the libraries with the actual developers via email lists.

1.2 Types of spatial data

You'll learn how to work with the two main types of spatial data, vector and raster. Vector data is made up of points, lines, and polygons, while raster data is a two- or three-dimensional array of data values such as the pixels in a photograph. A dataset containing country boundaries is an example of vector data. In this case, each country is generally represented as a polygon. Datasets that use lines to represent roads or rivers, or points to show the location of weather stations, are other examples. Early primitive maps, such as those drawn on cave walls, only showed the features themselves. Later maps contained labels for features of interest such as cities or seaports; for example, the Portolan map of northwest Africa shown in figure 1.1.

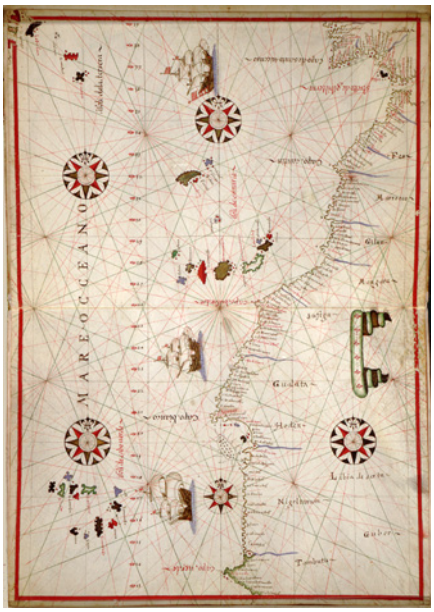


Figure 1.1 A Portolan map of the northwest coast of Africa, circa 1590

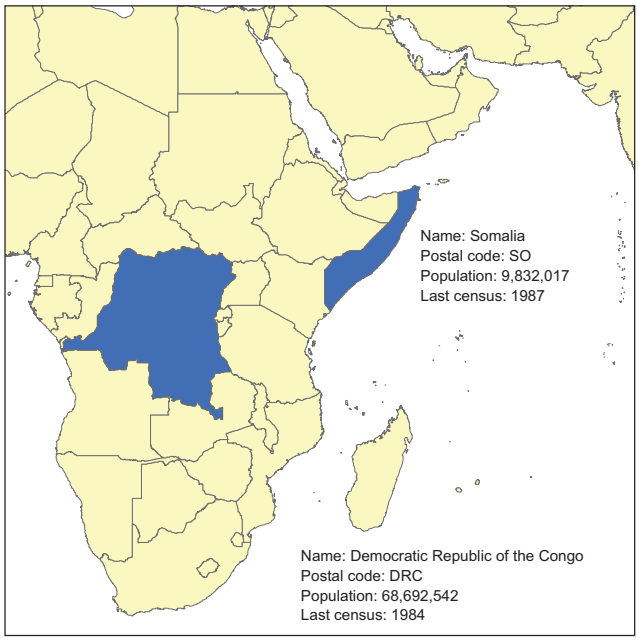
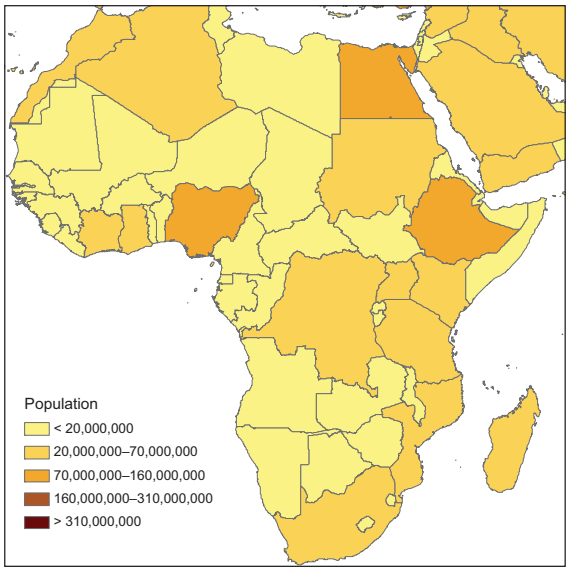


Figure 1.2 You can store attributes such as name and population for each geographic feature in a dataset.

Using digital data, you have the advantage of attaching multiple attribute values to each feature, whether you plan to display the information on a map or not. For each road, you can store information such as its name, speed limit, number of lanes, or anything else you can think of. Figure 1.2 shows an example of data you might store with each country in a dataset.



Of the several reasons why this is useful, the obvious one is that you can label features using one of the attributes. For example, figure 1.2 could show country names as well as outlines. All of this data can also help you make more-interesting maps that might even tell a story. The population counts stored for each feature in figure 1.2 could be used to symbolize countries based on population, so it's evident at a glance which countries are most populated (figure 1.3).

Figure 1.3 Countries symbolized based on population

Figure 1.4 Lake Victoria straddles Uganda, Kenya, and Tanzania. Spatial analysis could help you determine the proportion of the lake that falls in each country.

Spatial overlay analyses are also easy using vector data. Say you wanted to know what percentage of Lake Victoria was in Uganda, Kenya, and Tanzania. You could always guesstimate the answer based on figure 1.4, but you could also use GIS software to get more accurate numbers. You'll do simple analyses like this by the time you finish this book.

Attribute values attached to features can also add to the power of spatial operations. For example, say you had a dataset containing the locations of water wells with attributes that included depth and flow rate. If you also had a dataset for the same area containing geologic landforms or soil types, you could analyze this data to see if flow rate or required well depth was affected by landform or soil type.

Unlike the early mapmakers, you also have access to raster data. Rasters, as the datasets are called, are two- or three-dimensional arrays of values, the way a photograph is a two-dimensional array of pixel values. In fact, aerial photographs such as the one shown in figure 1.5 are a commonly used type of raster data. Satellite images sometimes look similar, although they generally have lower

Figure 1.5 An aerial photograph near Seattle, Washington

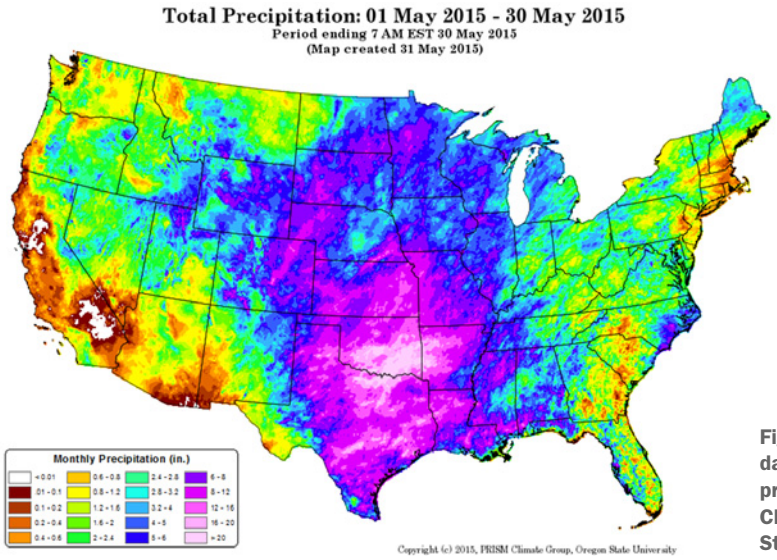


Figure 1.6 A raster dataset showing precipitation (PRISM Climate Group, Oregon State University, 2015)

resolutions. The cool thing about satellite imagery is that much of it is collected using nonvisible light so it can provide information that a simple photograph cannot.

Raster datasets are well suited to any continuous data, not only photographs. Precipitation data like that shown in figure 1.6 is a good example. Rain doesn't usually stop at a sudden boundary, so it's hard to draw a polygon around it. Instead, a grid of precipitation amounts works much better and can capture local variation more easily. The same idea applies to temperature data, and many other variables, as well. Another example is a *digital elevation model* (DEM), in which each pixel contains an elevation value.

Raster data is better suited for different types of analysis than vector data. Satellite imagery and aerial photos are commonly used for tasks such as vegetation mapping. Because water only flows downhill, elevation models can be used to determine watershed boundaries. Even simple math can be used to perform useful analyses with raster data. For example, simple ratios of one wavelength value to another can help identify healthy vegetation or measure soil moisture.

Blocks of adjacent pixels can also be used to calculate useful information. For example, you can use a DEM to calculate slope, which can then be used for runoff analysis, vegetation mapping, or planning a ski resort. But to calculate slope, you need the elevation of surrounding cells. In figure 1.7, you use all of the pixel values shown

54	53	51
53	52	50
50	50	48

Figure 1.7 All nine elevation values shown here would be used to calculate the slope for the center pixel.

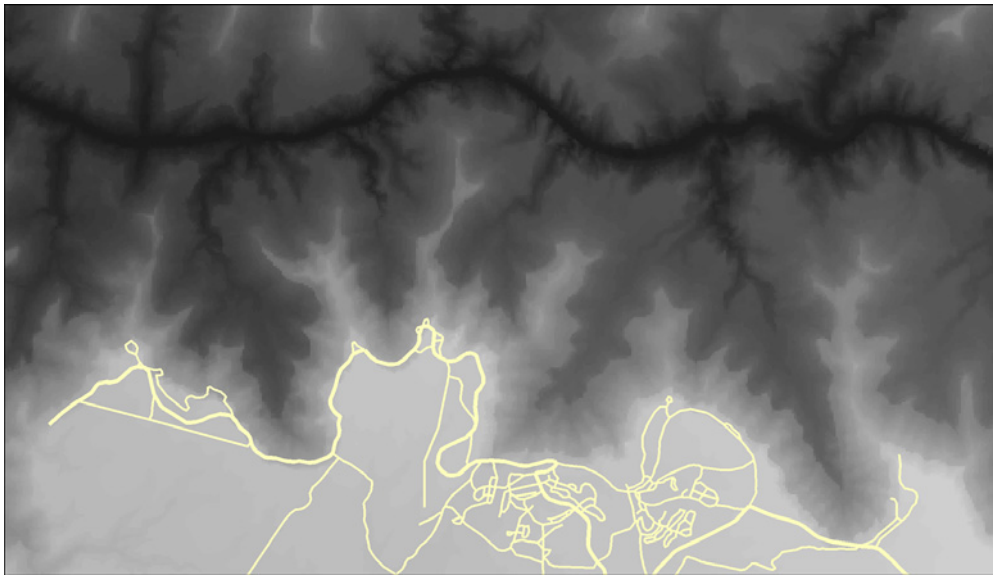


Figure 1.8 Simple map of the Grand Canyon with vector roads layer drawn on top of a raster elevation dataset

to calculate the slope of the center pixel. For any other pixel, you need the surrounding nine cells to calculate slope for it, too. These sets of pixels are called *windows*, and you can do many other kinds of analyses by moving a window around a raster so each pixel is in the center of its own window.

Vector and raster data can also be used together. Think of a hybrid web mapping application that shows a photographic basemap with roads drawn on top of it. The basemap is raster data and the roads shown on top are vectors. Figure 1.8 shows an example of a simple map that uses a raster DEM of the Grand Canyon as a basemap and shows a vector line dataset drawn on top.

1.3 What is geoprocessing?

Geoprocessing is a general term for manipulating spatial data, whether raster or vector. As you can imagine, that covers an awful lot of ground. I've always thought of using GIS with geoprocessing as a tool much like statistics in that it can be applied to pretty much everything. You even use geoprocessing in your daily life, whether you realize it or not. For example, I tend to take a different route to work depending on whether I'm driving or riding a bicycle because I prefer to avoid high-traffic roads with no shoulder when riding my bike. Steep hills are also not a concern while driving, but they are when I'm biking. Basing my route selection not only on spatial factors such as the direction of the road and elevation gain, but also on attributes such as the amount of traffic and road width is a type of geoprocessing. You probably make similar decisions every day.

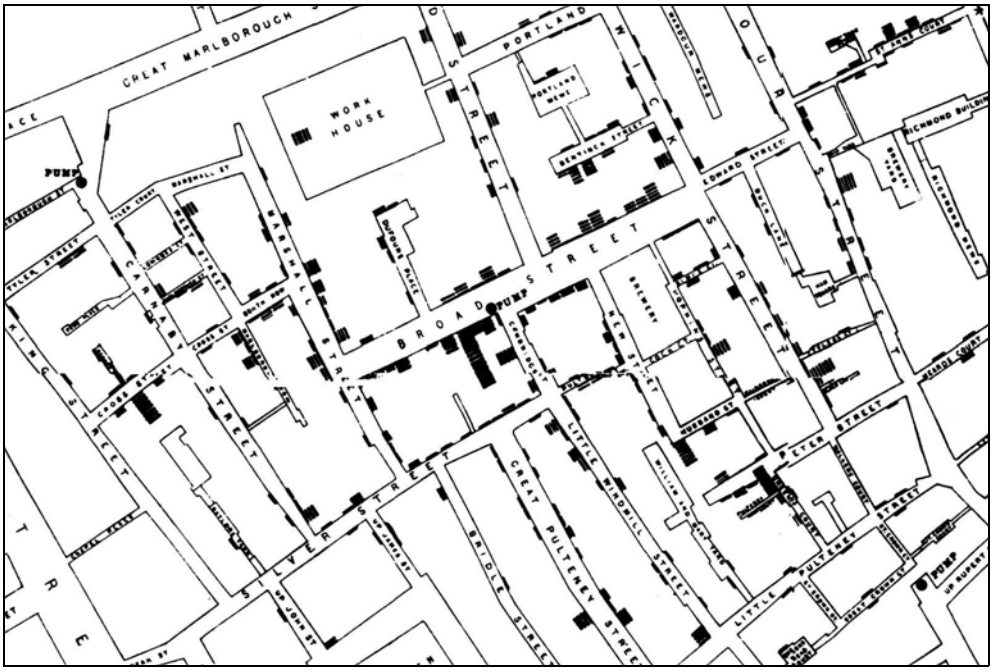


Figure 1.9 Part of John Snow's map of the Soho cholera outbreak of 1854

You have many reasons to be interested in geoprocessing, other than selecting a route to work. Let's look at a few examples of applications. One famous example of early spatial analysis is the story of John Snow, an English physician who lived in the 1800s. Although parts of the story have been disputed, the gist of it is that he used spatial analysis to determine the cause of a cholera outbreak in 1854. A section of his map is shown in figure 1.9, with the Broad Street pump in the middle. You can see that it looks like bar charts are anchored on nearby streets. Each of these bars is made of horizontal lines, with one per cholera victim. Snow realized that most of the victims probably got their water from the pump on Broad Street, because that was the closest one, and he convinced authorities to shut the pump down. This is significant not only because it's an early example of spatial analysis, but also because it wasn't yet known that cholera was contracted from contaminated water. Because of this, Snow is considered one of the fathers of modern epidemiology.

Spatial analysis is still an important part of epidemiology, but it's used for many other things, too. I've worked on projects that include studying the habits of a threatened species, modeling vegetation cover over large areas, comparing data from pre- and post-flood events to see how the river channels changed, and modeling carbon sequestration in forests. You can probably find examples of spatial analysis wherever your interests lie. Let's consider a few more examples.

Chinese researchers Luo et al.[1] used spatial analysis, along with historical records, to pinpoint the locations of missing courier stations along the Silk Road. The historical records contained descriptions of the route, including distance traveled and general direction between stations. The locations of several stations were already known, and the researchers knew that ancient travelers were unlikely to follow a straight line, but instead follow rivers or other landforms. They used all of this information to determine likely geographic areas for the still-missing stations. They then used high-resolution satellite imagery to search these areas for geometric shapes that could be station ruins. After visiting the sites in person, they determined that one, in fact, was an old courier station, and two others were likely military facilities during the Han Dynasty.

For a completely different application, Moody et al.[2] were interested in the potential for using microalgae as a biofuel. They used a microalgae growth model and meteorological data from various locations around the globe to simulate biomass productivity. Because the meteorological data was only from certain sites, the results were then spatially interpolated to provide a global map of productivity potential. It turns out that the most promising locations are in Australia, Brazil, Colombia, Egypt, Ethiopia, India, Kenya, and Saudi Arabia.

This is interesting, but spatial analyses also affect your everyday life. Have you noticed that your automobile insurance premium differs depending on where you live? It's likely that a sort of spatial analysis also affected the location of your favorite coffee shop or grocery store. Several new elementary and high schools are being built in my community, and their locations were determined in part by the spatial distribution of future students, along with the availability of suitable pieces of real estate.

Spatial analysis isn't limited to geography, either. Rose et al.[3] demonstrated that GIS can be used to analyze the distribution of nano- and microstructures in bone. They could use this to see how bone remodeling events corresponded to parts of the bone that experience high levels of compression and tension.

You personally might need to make data more suitable for a map, such as eliminating unwanted features or simplifying complex lines so they display faster on a web map. Or you might analyze demographic data to plan for future transportation needs. Perhaps you're interested in how vegetation responds to different land management practices, such as prescribed burns or mowing. Or maybe it's something else entirely.

Although geoprocessing techniques can be rather complicated, many are fairly simple. It's the simple ones that you'll learn about in this book, but they're the foundation for everything else. By the time you're done, you'll read and write spatial data in many

---

[1] Luo, L., X. Wang, C. Liu, H. Guo, and X. Du. 2014. Integrated RS, GIS and GPS approaches to archaeological prospecting in the Hexi Corridor, NW China: a case study of the royal road to ancient Dunhuang. Journal of Archaeological Science. 50: 178-190. doi:10.1016/j.jas.2014.07.009.

[2] Moody, J. W., C. M. McGinty, and J. C. Quinn. 2014. Global evaluation of biofuel potential from microalgae. Proceedings of the National Academy of Sciences of the United States of America. 111: 8691-8696. doi: 10.1073/pnas.1321652111.

[3] Rose, D. C., A. M. Agnew, T. P. Gocha, S. D. Stout, and J. S. Field. 2012. Technical note: The use of geographical information systems software for the spatial analysis of bone microstructure. American Journal of Physical Anthropology. 148: 648–654. doi: 10.1002/ajpa.22099.

formats, both vector and raster. You'll subset vector data by attribute value or by spatial location. You'll know how to perform simple vector geoprocessing, including overlay and proximity analyses. In addition, you'll know how to work with raster datasets, including resizing pixels, performing calculations based on multiple datasets, and moving window analyses.

You'll know how to do all of this with Python rather than by pushing buttons in a software package. The ability to script your processes like this is extremely powerful. Not only does it make it easy to batch process many datasets at once (something I do often), but it gives you the ability to customize your analysis instead of being limited to what the software user interface allows. You can build your own custom toolkits based on your workflow, and use these over and over. Automation is another big one, and it's the reason I fell in love with scripting in the first place. I hate pushing buttons and doing the same thing over and over, but I'll happily spend time figuring out how to automate something so I never have to think about it again. One last advantage that I'll mention here is that you always know exactly what you did, as long as you don't lose your script, because everything is right there.

1.4 Exploring your data

You'll see ways to visualize your data as you work with it in Python, but the best way to explore the data is still to use a desktop GIS package. It allows you to easily visualize the data spatially in multiple ways, but also inspect the attributes included with the data. If you don't have access to GIS software already, QGIS is a good open source option and is the one we'll be using when needed in this book. It's available from www.qgis.org, and it runs on Linux, Mac OS X, and Windows.

Downloadable code and sample data

The examples in this book use code and sample data that's available for download from the following links. You'll need to download these if you want to follow along. The code contains examples from the book but also custom utilities used by the examples, and all of the data used in the examples is included.

- Code: https://github.com/cgarrard/osgeopy-code and www.manning.com/books/geoprocessing-with-python
- Data: https://app.box.com/osgeopy and www.manning.com/books/geoprocessing-with-python

This isn't a book on QGIS, so I won't talk much about how to use it. Documentation is available on their website, and you can find one or two books published on the topic. However, I'll briefly discuss how to load data and take a look. If you've never used a GIS before, then QGIS might look a bit daunting when you first open it up, but it's not hard to use it to view data. For example, to load up one of the shapefiles in the example data for this book, select Add Vector Layer… from the Layer menu in QGIS. In the dialog that opens, make sure that the File button is selected and then use the Browse

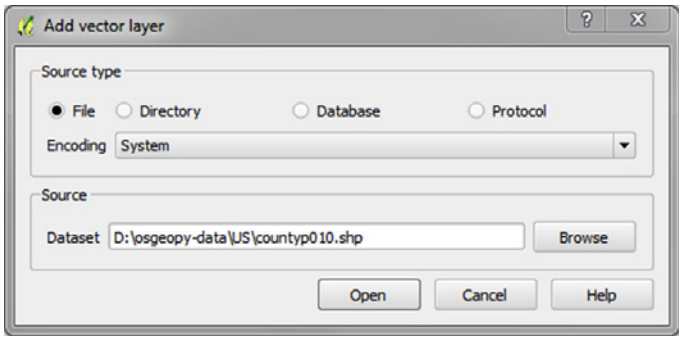


Figure 1.10 The dialog for adding a vector layer to QGIS

button to select a shapefile. A good choice to start out with is the countyp010.shp file in the US folder (figure 1.10).

After selecting a file, click Open in the Add vector layer dialog, and the spatial data will draw in QGIS, as shown in figure 1.11. You can use the magnifying glass tool (circled in figure 1.11) to zoom in on part of the map.

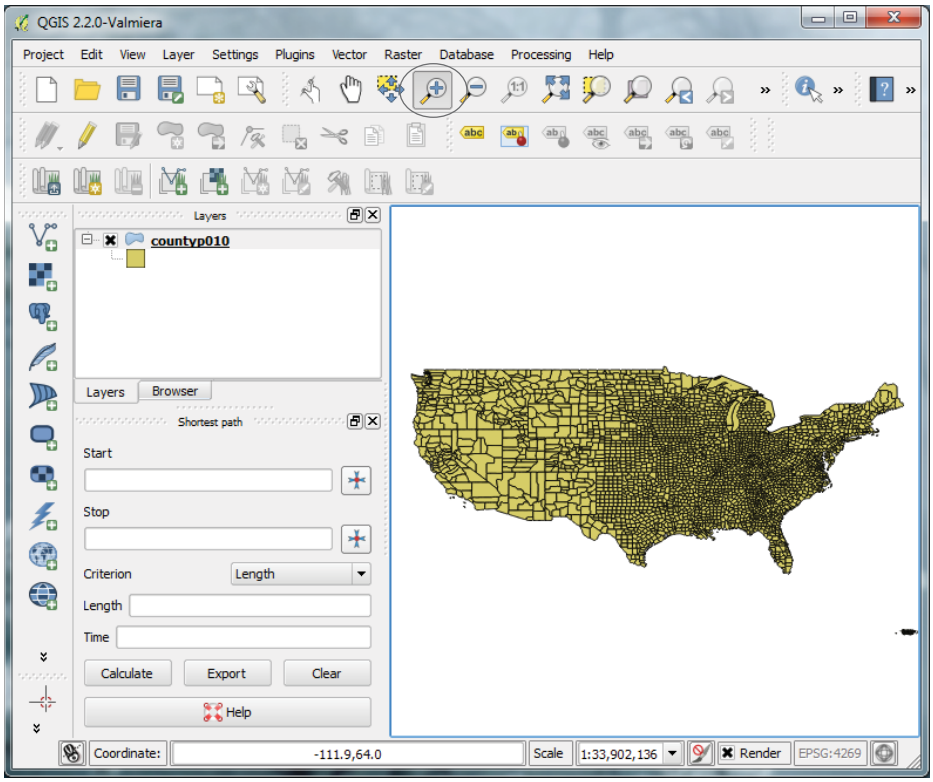


Figure 1.11 QGIS window immediately after loading countyp010.shp

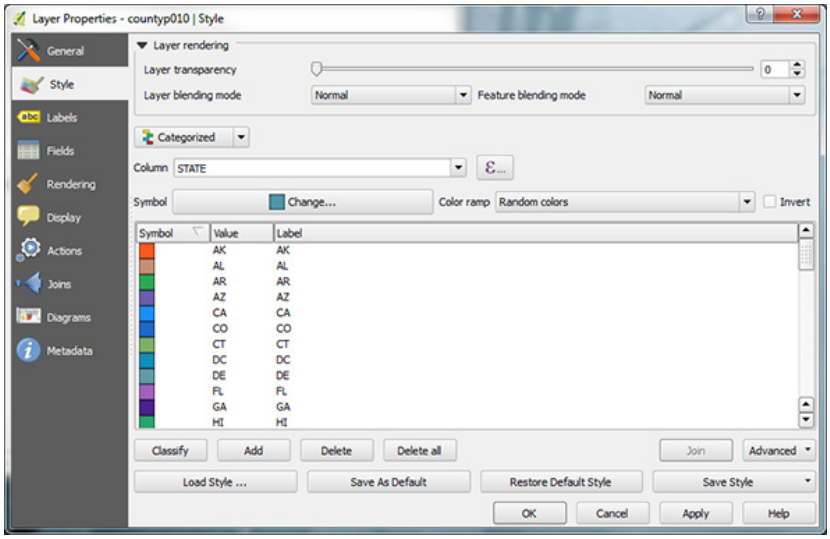


Figure 1.12 QGIS Style dialog configured to draw the counties in each state in a different color

You'll also see the name of the layer, countyp010 in this case, shown in the Layers list on the left. Double-click on a layer and you'll get a Properties dialog. If you click on the Style tab, then you can change how the data is drawn. Let's change the counties layer so that the counties are not all drawn with the same color, but instead the color depends on the state the county is in. To do this, choose Categorized from the dropdown list, set the column to STATE, select a Color ramp from the dropdown list, and then click Classify. You'll see a list of all of the states and the colors they'll be drawn with, as shown in figure 1.12. You can change the color ramp by selecting a new one from the list, clicking Delete All, and then clicking Classify again. You can also change a particular entry in the list by double-clicking on the color swatch next to the state abbreviation.

> **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.

Once you're happy with your colors, click Apply, and the colors will be applied in the main QGIS window (figure 1.13).

You can view the attribute data that's attached to the spatial data by right-clicking on the layer name in the Layers list and selecting Open Attribute Table. Each row in the table shown in figure 1.14 corresponds to a county drawn on the map. In fact, try selecting a row by clicking on the number in the left-most column and then clicking on the Zoom map to selected rows button (circled in figure 1.14) and watch what happens.

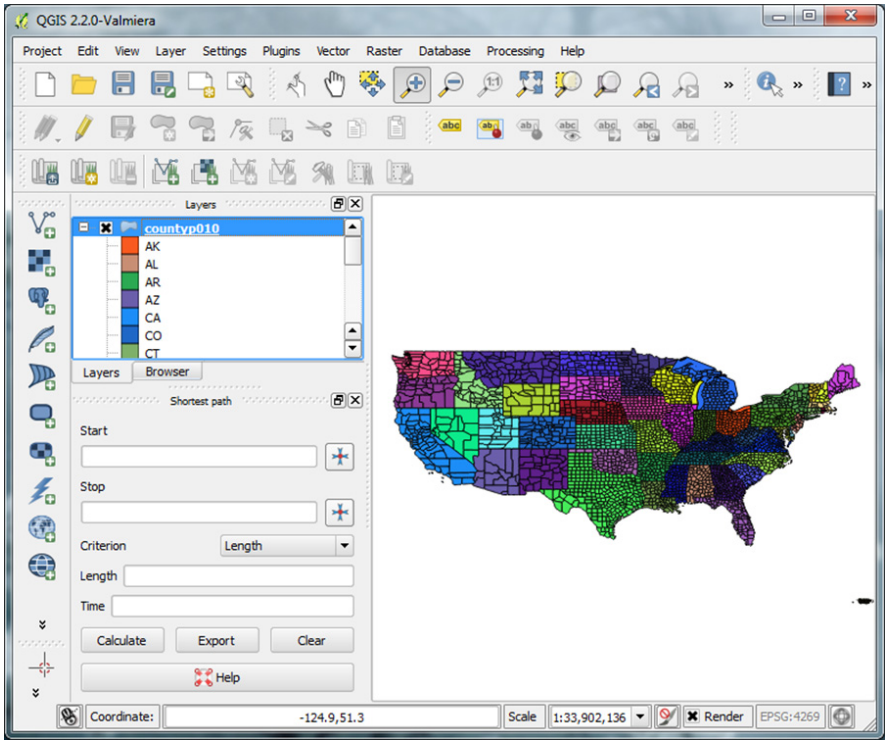
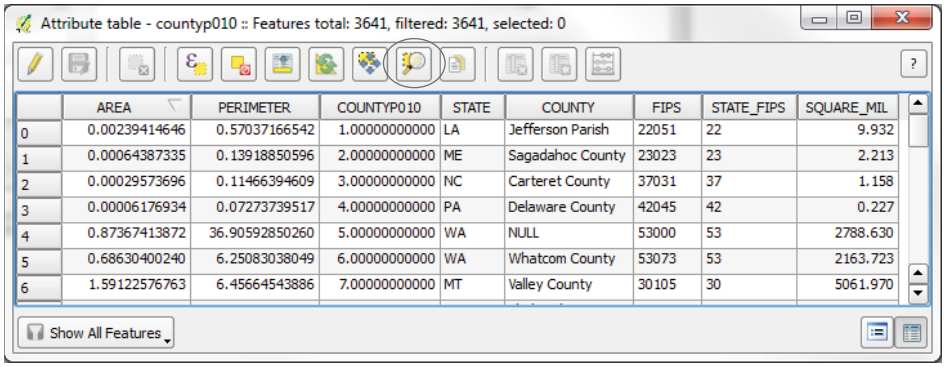


Figure 1.13 Results of applying the symbology from figure 1.12 to the counties layer

.



	AREA	PERIMETER	COUNTYP010	STATE	COUNTY	FIPS	STATE_FIPS	SQUARE_MIL
0	0.00239414646	0.57037166542	1.00000000000	LA	Jefferson Parish	22051	22	9.932
1	0.00064387335	0.13918850596	2.00000000000	ME	Sagadahoc County	23023	23	2.213
2	0.00029573696	0.11466394609	3.00000000000	NC	Carteret County	37031	37	1.158
3	0.00006176934	0.07273739517	4.00000000000	PA	Delaware County	42045	42	0.227
4	0.87367413872	36.90592850260	5.00000000000	WA	NULL	53000	53	2788.630
5	0.68630400240	6.25083038049	6.00000000000	WA	Whatcom County	53073	53	2163.723
6	1.59122576763	6.45664543886	7.00000000000	MT	Valley County	30105	30	5061.970

Figure 1.14 Attribute table for the counties layer

Take time to play with QGIS and read at least part of the documentation on the website. The software is extremely powerful and worth getting to know. I'll talk about it more throughout the book, but not a whole lot. You'll want to use it to inspect the sample data and the results of any data you create, however.

1.5 Summary

- Python is a powerful multiplatform programming language that's relatively easy to learn.
- Free and open source software is not only free with regard to price (free beer), but also allows for many freedoms with how it's used (free speech).
- Many excellent open source Python modules exist for processing both vector and raster geospatial data.
- You don't give up quality by using open source tools. In fact several of these packages are also used by proprietary software.

2 Python basics

This chapter covers

- Using the Python interpreter vs. writing scripts
- Using the core Python data types
- Controlling the order of code execution

You can do many things with desktop GIS software such as QGIS, but if you work with spatial data for long, you'll inevitably want to do something that isn't available through the software's interface. If you know how to program, and are clever enough, you can write code that does exactly what you need. Another common scenario is the need to automate a repetitive processing task instead of using the point-and-click method over and over again. Not only is coding more fun and intellectually stimulating than pointing and clicking, but it's also much more efficient when it comes to repetitive tasks. You have no shortage of languages you could learn and work with, but because Python is used with many GIS software packages, including QGIS and ArcGIS, it's an excellent language for working with spatial data. It's also powerful, but at the same time a relatively easy-to-learn language, so that makes it a good choice if you're starting out with programming.

Another reason for using Python is that it's an interpreted language, so programs written in Python will run on any computer with an interpreter, and

interpreters exist for any operating system you're likely to use. To run a Python script, you need the script and an interpreter, which is different from running an .exe file, for example, where you only need one file. But if you have an .exe file, you can only run it under the Windows operating system, which is a bummer if you want to run it on a Mac or Linux. However, if you have a Python script, you can run it anywhere that has an interpreter, so you're no longer limited to a single operating system.

2.1 Writing and executing code

Another advantage of interpreted languages is that you can use them interactively. This is great for playing around and learning a language, because you can type a line of code and see the results instantly. You can run the Python interpreter in a terminal window, but it's probably easier to use *IDLE*, which is a simple development environment installed with Python. Two different types of windows exist in IDLE, *shells* and *edit windows*. A shell is an interactive window in which you can type Python code and get immediate results. You'll know that you're looking at an interactive window if you see a `>>>` prompt, like that in figure 2.1. You can type code after this prompt and execute it by pressing Enter. Many of the examples in this book are run this way to show results. This is an inefficient way to run more than a few lines of code, and it doesn't save your code for later use. This is where the edit window comes in. You can use the File menu in IDLE to open a new window, which will contain an empty file. You can type your code in there and then execute the script using the Run menu, although you'll need to save it with a .py extension first. The output from the script will be sent to the interactive window. Speaking of output, in many of the interactive examples in this book I type a variable name to see what the variable contains, but this won't work if you're running the code from a script. Instead, you need to use `print` to explicitly tell it to send information to the output window.

In figure 2.1 the string I typed, `'Hello world!'`, and the output are color coded. This syntax highlighting is useful because it helps you pick out keywords, built-in functions, strings, and error messages at a glance. It can also help you find spelling mistakes if something doesn't change color when you expect it to. Another useful feature of IDLE is *tab completion*. If you start typing a variable or function name and then press the Tab

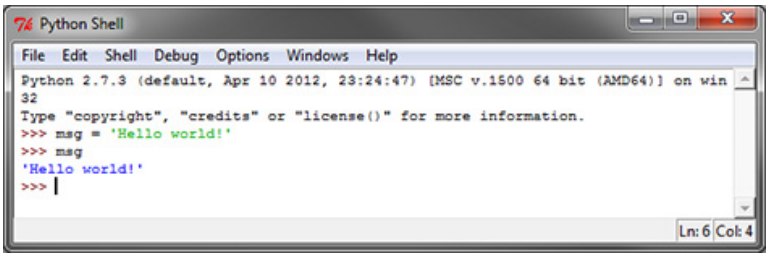


Figure 2.1 An IDLE shell window

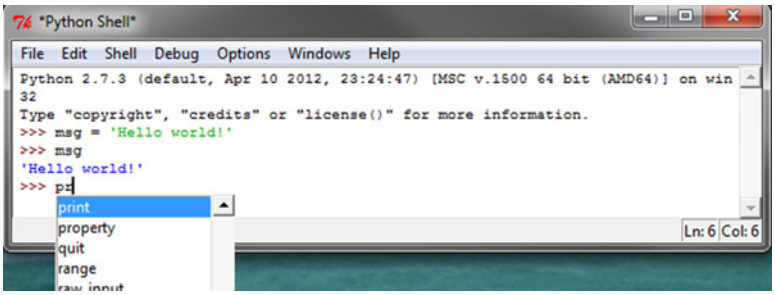


Figure 2.2 Start typing and press the Tab key in order to get a list of possible variables or functions that match what you were typing.

key, a list of options will pop up, as shown in figure 2.2. You can keep typing, and it will narrow the search. You can also use arrow keys to scroll through the list. When the word you want is highlighted, press Tab again, and the word will appear on your screen.

Because Python scripts are plain text files, you aren't forced to use IDLE if you don't want to. You can write scripts in whatever text editor you prefer. Many editors are easy to configure, so you can run a Python script directly without leaving the editor. See the documentation for your favorite editor to learn how to do this. Packages that are designed specifically for working with Python code are Spyder, PyCharm, Wing IDE, and PyScripter. Everybody has their own favorite development environment, and you may need to play with a few different ones before you find an environment that you like.

2.2 Basic structure of a script

Some of the first things you'll see right at the top of most Python scripts are `import` statements. These lines of code load additional modules so that the scripts can use them. A module is basically a library of code that you can access and use from your scripts, and the large ecosystem of specialized modules is another advantage to using Python. You'd have a difficult time working with GIS data in Python without extra modules that are designed for this, similar to the way tools such as GIMP and Photoshop make it easier to work with digital images. The whole point of this book is to teach you how to use these tools for working with GIS data. Along the way, you'll also use several of the modules that come with Python because they're indispensable for tasks such as working with the file system.

Let's look at a simple example that uses one of the built-in modules. The first thing you need to do to use a module is load it using `import`. Then you can access objects in the module by prefixing them with the module name so that Python knows where to find them. This example loads the `random` module and then uses the `gauss` function contained in that module to get a random number from the standard normal distribution:

```
>>> import random
>>> random.gauss(0, 1)
-0.22186423850882403
```

Another thing you might notice in a Python script is the lack of semicolons and curly braces, which are commonly used in other languages for ending lines and setting off blocks of code. Python uses whitespace to do these things. Instead of using a semicolon to end a line, press Enter and start a new line. Sometimes one line of code is too long to fit comfortably on one line in your file, however. In this case, break your line at a sensible place, such as right after a comma, and the Python interpreter will know that the lines belong together. As for the missing curly braces, Python uses indentation to define blocks of code instead. This may seem weird at first if you're used to using braces or `end` statements, but indentation works as well and forces you to write more readable code. Because of this, you need to be careful with your indentations. In fact, it's common for beginners to run into syntax errors because of wayward indentations. For example, even an extra space at the beginning of a line of code will cause an error. You'll see examples of how indentation is used in section 2.5.

Python is also case sensitive, which means that uppercase and lowercase letters are different from one another. For example, `random.Gauss(0, 1)` wouldn't have worked in the last example because `gauss` needs to be all lowercase. If you get error messages about something being undefined (which means Python doesn't know what it is), but you're sure that it exists, check both your spelling and your capitalization for mistakes.

It's also a good idea to add comments to your code to help you remember what it does or why you did it a certain way. I can guarantee that things that are obvious as you're writing your code will not be so obvious six months later. Comments are ignored by Python when the script is run, but can be invaluable to the real people looking at the code, whether it's you or someone else trying to understand your code. To create a comment, prefix text with a hash sign:

```
# This is a comment
```

In addition to comments, descriptive variable names improve the legibility of your code. For example, if you name a variable `m`, you need to read through the code to figure out what's stored in that variable. If you name it `mean_value` instead, the contents will be obvious.

2.3 Variables

Unless your script is extremely simple, it will need a way to store information as it runs, and this is where variables come in. Think about what happens when you use software to open a file, no matter what kind of file it is. The software displays an Open dialog, you select a file and click OK, and then the file is opened. When you press OK, the name of the selected file is stored as a variable so that the software knows what file to open. Even if you've never programmed anything in your life, you're probably familiar with this concept in the mathematical sense. Think back to algebra class and computing the value of y based on the value of x. The x variable can take on any value, and y changes in response. A similar concept applies in programming. You'll use many different variables, or x's, that will affect the outcome of your script. The outcome can be anything you want it to be and isn't limited to a single y value, however. It might be

a number, if your goal is to calculate a statistic on your data, but it could as easily be one or more entirely new datasets.

Creating a variable in Python is easy. Give it a name and a value. For example, this assigns the value of `10` to a variable called `n` and then prints it out:

```
>>> n = 10
>>> n
10
```

If you've used other programming languages such as C++ or Java, you might be wondering why you didn't need to specify that the variable `n` was going to hold an integer value. Python is a dynamically typed language, which means that variable types aren't checked until runtime, and you can even change the data type stored in a variable. For example, you can switch `n` from an integer to a string and nobody will complain:

```
>>> n = 'Hello world'
>>> n
Hello world
```

Although you can store whatever you want in a variable without worrying about data type, you will run into trouble if you try to use the variable in a way that's inconsistent with the kind of data stored in it. Because the data types aren't checked until runtime, the error won't happen until that line of the script is executed, so you won't get any warning beforehand. You'll get the same errors in the Python interactive window that would occur in a script, so you can always test examples there if you're not sure if something will work. For example, you can't add strings and integers together, and this shows what happens if you try:

```
>>> msg = n + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

Remember that `n` contains `Hello world`, which cannot be added to `1`. If you're using Python 2.7, the core of the problem is the same, but your error message will look like this instead:

```
TypeError: cannot concatenate 'str' and 'int' objects
```

Notice that you use a single equal sign to assign a value to a variable. To test for equality, always use a double equal sign:

```
>>> n = 10
>>> n == 10
True
```

When you're first starting out, you might be more comfortable hardcoding values into your script instead of using variables when you don't have to. For example, say you need to open a file in the script, maybe on line 37. You'll probably be tempted to type the filename on line 37 when the file is opened. This will certainly work, but you'll find that things are easier to change later if you instead define a variable containing

the filename early in the script and then use that variable on line 37. First, this makes it easier to find the values you need to change, but even more importantly, it will be much easier to adapt your code so that you can use it in more situations. Instead of line 37 looking something like this,

```
myfile = open('d:/temp/cities.csv')
```

you'd define a variable early on and then use it when needed:

```
fn = 'd:/temp/cities.csv'
<snip a bunch of code>
myfile = open(fn)
```

It might be hard to remember to do this at first, but you'll be glad you did if you have to adapt your code to use other data.

2.4 Data types

As your code becomes more complex, you'll find that it's extremely difficult to store all of the information that your script needs as numbers and strings. Fortunately, you can use many different types of data structures, ranging from simple numbers to complex objects that can contain many different types of data themselves. Although an infinite number of these object types can be used (because you can define your own), only a small number of core data types exist from which the more complex ones are built. I'll briefly discuss several of those here. Please see a more comprehensive set of Python documentation for more details, because this leaves out much information.

2.4.1 Booleans

A *Boolean* variable denotes true or false values. Two case-sensitive keywords, `True` and `False`, are used to denote these values. They can be used in standard Boolean operations, like these:

```
>>> True or False
True
>>> not False
True
>>> True and False
False
>>> True and not False
True
```

Other values can also resolve to `True` or `False` when value testing and performing Boolean operations. For example, `0`, the `None` keyword, blank strings, and empty lists, tuples, sets, and dictionaries all resolve to `False` when used in Boolean expressions. Anything else resolves to `True`. You'll see examples of this in section 2.5.

2.4.2 Numeric types

As you'd expect, you can use Python to work with numbers. What you might not expect, however, is that distinct kinds of numbers exist. *Integers* are whole numbers,

such as 5, 27, or 592. *Floating-point numbers*, on the other hand, are numbers with decimal points, such as 5.3, 27.0, or 592.8. Would it surprise you to know that 27 and 27.0 are different? For one, they might take up different amounts of memory, although the details depend on your operating system and version of Python. If you're using Python 2.7 there's a major difference in how the two numbers are used for mathematical operations, because integers don't take decimal places into account. Take a look at this Python 2.7 example:

```
>>> 27 / 7
3
>>> 27.0 / 7.0
3.857142857142857
>>> 27 / 7.0
3.857142857142857
```

As you can see, if you divide an integer by another integer, you still end up with an integer, even if there's a remainder. You get the correct answer if one or both of the numbers being used in the operation is floating-point. This behavior has changed in Python 3.x, however. Now you get floating-point math either way, but you can still force integer math using the // floor division operator:

```
>>> 27 / 7
3.857142857142857
>>> 27 // 7
3
```

WARNING Python 3.x performs floating-point math by default, even on integers, but older versions of Python perform integer math if all inputs are integers. This integer math often leads to undesirable results, such as 2 instead of 2.4, in which case you must ensure that at least one input is floating-point.

Fortunately, you have a simple way to convert one numeric data type to the other, although be aware that converting floating-point to integer this way truncates the number instead of rounding it:

```
>>> float(27)
27.0
>>> int(27.9)
27
```

If you want to round the number instead, you must use the `round` function:

```
>>> round(27.9)
28
```

Python also supports complex numbers, which contain real and imaginary parts. As you might recall, these values result when you take the square root of a negative number. We won't use complex numbers in this book, but you can read more about them at python.org if you're interested.

2.4.3 Strings

Strings are text values, such as `'Hello world'`. You create a string by surrounding the text with either single or double quotes—it doesn't matter which, although if you start a string with one type, you can't end it with the other because Python won't recognize it as the end of the string. The fact that either one works makes it easy to include quotes as part of your string. For example, if you need single quotes inside your string, as you would in a SQL statement, surround the entire string with double quotes, like this:

```
sql = "SELECT * FROM cities WHERE country = 'Canada'"
```

If you need to include the same type of quote in your string that you're using to delineate it, you can use a backslash before the quote. The first example here results in an error because the single quote in "don't" ends the string, which isn't what you want. The second one works, thanks to the backslash:

```
>>> 'Don't panic!'
  File "<stdin>", line 1
    'Don't panic!'
         ^
SyntaxError: invalid syntax
>>> 'Don\'t panic!'
"Don't panic!"
```

Notice the caret symbol (`^`) under the spot where Python ran into trouble. This can help you narrow down where your syntax error is. The double quotes that surround the string when it's printed aren't part of the string. They show that it's a string, which is obvious in this case, but wouldn't be if the string was `"42"` instead. If you use the `print` function, the quotes aren't shown:

```
>>> print('Don\'t panic!')
Don't panic!
```

> **TIP** Although most of these examples from the interactive window don't use `print` to send output to the screen, you must use it to send output to the screen from a script. If you don't, it won't show up. In Python 3, `print` is a function and like all functions, you must pass the parameters inside parentheses. In Python 2, `print` is a statement and the parentheses aren't required, but they won't break anything, either.

JOINING STRINGS

You have several ways to join strings together. If you're only concatenating two strings, then the simplest and fastest is to use the `+` operator:

```
>>> 'Beam me up ' + 'Scotty'
'Beam me up Scotty'
```

If you're joining multiple strings, the `format` method is a better choice. It can also join values together that aren't all strings, something the `+` operator can't do. To use it, you create a template string that uses curly braces as placeholders, and then pass values to take the place of the placeholders. You can read the Python documentation online to

see the many ways you can use this for sophisticated formatting, but we'll look at the basic method of specifying order. Here, the first item passed to `format` replaces the `{0}` placeholder, the second replaces `{1}`, and so on:

```
>>> 'I wish I were as smart as {0} {1}'.format('Albert', 'Einstein')
'I wish I were as smart as Albert Einstein'
```

To see that the numeric placeholders make a difference, try switching them around but leaving everything else the same:

```
>>> 'I wish I were as smart as {1}, {0}'.format('Albert', 'Einstein')
'I wish I were as smart as Einstein, Albert'
```

The fact that the placeholders reference specific values means that you can use the same placeholder in multiple locations if you need to insert an item in the string more than once. This way you don't have to repeat anything in the list of values passed to `format`.

ESCAPE CHARACTERS

Remember the backslash that you used to include a quote inside a string earlier? That's called an *escape character* and can also be used to include nonprintable characters in strings. For example, `"\n"` includes a new line, and `"\t"` represents a tab:

```
>>> print('Title:\tMoby Dick\nAuthor:\tHerman Melville')
Title:  Moby Dick
Author: Herman Melville
```

The fact that Windows uses backslashes as path separators causes angst for beginning programmers who use Windows, because they tend to forget that a single backslash isn't a backslash. For example, pretend you have a file called cities.csv in your d:\temp folder. Try asking Python if it exists:

```
>>> import os
>>> os.path.exists('d:\temp\cities.csv')
False
```

To get an idea of why that fails, when you know that the file does indeed exist, try printing the string instead:

```
>>> print('d:\temp\cities.csv')
d:      emp\cities.csv
```

The `"\t"` was treated as a tab character! You have three ways to solve this problem. Either use forward slashes or double backslashes, or prefix the string with an `r` to tell Python to ignore escape characters:

```
>>> os.path.exists('d:/temp/cities.csv')
True
>>> os.path.exists('d:\\temp\\cities.csv')
True
>>> os.path.exists(r'd:\temp\cities.csv')
True
```

I prefer the latter method if I'm copying and pasting paths, because it's much easier to add one character at the beginning than to add multiple backslashes.

2.4.4 Lists and tuples

A *list* is an ordered collection of items that are accessed via their index. The first item in the list has index 0, the second has index 1, and so on. The items don't even have to all be the same data type. You can create an empty list with a set of square brackets, `[]`, or you can populate it right off the bat. For example, this creates a list with a mixture of numbers and strings and then accesses some of them:

```
>>> data = [5, 'Bob', 'yellow', -43, 'cat']
>>> data[0]
5
>>> data[2]
'yellow'
```

You can also use offsets from the end of the list, with the last item having index -1:

```
>>> data[-1]
'cat'
>>> data[-3]
'yellow'
```

You're not limited to retrieving one item at a time, either. You can provide a starting and ending index to extract a slice, or sublist. The item at the ending index isn't included in the returned value, however:

```
>>> data[1:3]
['Bob', 'yellow']
>>> data[-4:-1]
['Bob', 'yellow', -43]
```

You can change single values in the list, or even slices, using indices:

```
>>> data[2] = 'red'
>>> data
[5, 'Bob', 'red', -43, 'cat']
>>> data[0:2] = [2, 'Mary']
>>> data
[2, 'Mary', 'red', -43, 'cat']
```

Use `append` to add an item to the end of the list, and `del` to remove an item:

```
>>> data.append('dog')
>>> data
[2, 'Mary', 'red', -43, 'cat', 'dog']
>>> del data[1]
>>> data
[2, 'red', -43, 'cat', 'dog']
```

It's also easy to find out how many items are in a list or if it contains a specific value:

```
>>> len(data)
5
>>> 2 in data
True
>>> 'Mary' in data
False
```

Tuples are also ordered collections of items, but they can't be changed once created. Instead of brackets, tuples are surrounded by parentheses. You can access items and test for existence the same as with lists:

```
>>> data = (5, 'Bob', 'yellow', -43, 'cat')
>>> data[1:3]
('Bob', 'yellow')
>>> len(data)
5
>>> 'Bob' in data
True
```

Like I said, you're not allowed to change a tuple once it has been created:

```
>>> data[0] = 10
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Because of this, use lists instead of tuples when it's possible that the data will change.

Error messages are your friend

When you get an error message, be sure to look carefully at the information it provides because this can save you time figuring out the problem. The last line is a message giving you a general idea of what the problem is, as seen here:

```
>>> data[0] = 10
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

You could deduce from this error message that your code tried to edit a tuple object somehow. Before the error message, you'll see a list of the lines of code that were executed before it ran into a problem. This is called a *stack trace*. In this example, `<stdin>` means the interactive window, so the line number isn't as helpful. But look at the following, which traces through two lines of code:

```
Traceback (most recent call last):
  File "D:\Temp\trace_example.py", line 7, in <module>
    y = add(x, '1')
  File "D:\Temp\trace_example.py", line 2, in add
    return n1 + n2
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Code on line 7

Code on line 2

The last line tells you the error is from trying to add an integer and a string together. The trace tells you that the problem started with line 7 of the file trace_example.py. Line 7 calls a function called `add`, and the error happens on line 2 inside of that function. You can use the information from the stack trace to determine where the error occurred, and where the original line of code that triggered it is. In this example, you know that either you passed bad data to the `add` function on line 7, or else an error exists in the `add` function on line 2. That gives you two specific places to look for a mistake.

2.4.5 Sets

Sets are unordered collections of items, but each value can only occur once, which makes it an easy way to remove duplicates from a list. For example, this set is created using a list that contains two instances of the number 13, but only one is in the resulting set:

```
>>> data = set(['book', 6, 13, 13, 'movie'])
>>> data
{'movie', 6, 'book', 13}
```

You can add new values, but they'll be ignored if they're already in the set, such as `'movie'` in this example:

```
>>> data.add('movie')
>>> data.add('game')
>>> data
{'movie', 'game', 6, 'book', 13}
```

Sets aren't ordered, so you can't access specific elements. You can check if items are in the set, however:

```
>>> 13 in data
True
```

Sets also make it easy to do things such as combine collections (`union`) or find out which items are contained in both sets (`intersection`):

```
>>> info = set(['6', 6, 'game', 42])
>>> data.union(info)
{6, 'movie', 13, 'game', 'book', '6', 42}
>>> data.intersection(info)
{'game', 6}
```

All values from both sets

Only values contained in both

You've already seen that you can use sets to remove duplicates from a list. An easy way to determine if a list contains duplicate values is to create a set from the list and check to see if the set and list have the same length. If they don't, then you know duplicates were in the list.

2.4.6 Dictionaries

Dictionaries are indexed collections, like lists and tuples, except that the indices aren't offsets like they are in lists. Instead, you get to choose the index value, called a *key*. Keys can be numbers, strings, or other data types, as can the values they reference. Use curly braces to create a new dictionary:

```
>>> data = {'color': 'red', 'lucky number': 42, 1: 'one'}
>>> data
{1: 'one', 'lucky number': 42, 'color': 'red'}
>>> data[1]
'one'
```

```
>>> data['lucky number']
42
```

As with lists, you can add, change, and remove items:

```
>>> data[5] = 'candy'
>>> data
{1: 'one', 'lucky number': 42, 5: 'candy', 'color': 'red'}
>>> data['color'] = 'green'
>>> data
{1: 'one', 'lucky number': 42, 5: 'candy', 'color': 'green'}
>>> del data[1]
>>> data
{'lucky number': 42, 5: 'candy', 'color': 'green'}
```

You can also test to see if a key exists in the dictionary:

```
>>> 'color' in data
True
```

This is a powerful way to store data when you don't know beforehand what it will be. For example, say you needed to remember the spatial extent for each file in a collection of geographic datasets, but the list of datasets changed each time you ran your script. You could create a dictionary and use the filenames as keys and the spatial extents as values, and then this information would be readily available later in your script.

2.5 Control flow

The first script you write will probably consist of a sequence of statements that are executed in order, like all of the examples we have looked at so far. The real power of programming, however, is the ability to change what happens based on different conditions. Similar to the way you might use sale prices to decide which veggies to buy at the supermarket, your code should use data, such as whether it's working with a point or a line, to determine exactly what needs to be done. *Control flow* is the concept of changing this order of code execution.

2.5.1 If statements

Perhaps the simplest way to change execution order is to test a condition and do something different depending on the outcome of the test. This can be done with an `if` statement. Here's a simple example:

```
if n == 1:
    print('n equals 1')
else:
    print('n does not equal 1')
```

If the value of the `n` variable is `1`, then the string "n equals 1" will be printed. Otherwise, the string "n does not equal 1" will be printed. Notice that the `if` and `else` lines end with a colon and that the code depending on a condition is indented under the

condition. This is a requirement. Once you quit indenting code, then the code quits depending on the condition. What do you think the following code will print?

```
n = 1
if n == 1:
    print('n equals 1')
else:
    print('n does not equal 1')
print('This is not part of the condition')
```

Well, `n` is equal to `1`, so the equality message prints out, and then control is transferred to the first line of code that isn't indented, so this is the result:

```
n equals 1
This is not part of the condition
```

You can also test multiple conditions like this:

```
if n == 1:
    print('n equals 1')
elif n == 3:
    print('n equals 3')
elif n > 5:
    print('n is greater than 5')
else:
    print('what is n?')
```

In this case, `n` is first compared to `1`. If it's not equal to `1`, then it's compared to `3`. If it's not equal to that, either, then it checks to see if `n` is greater than `5`. If none of those conditions are true, then the code under the `else` statement is executed. You can have as many `elif` statements as you want, but only one `if` and no more than one `else`. Similar to the way the `elif` statements aren't required, neither is an `else` statement. You can use an `if` statement all by itself if you'd like.

This is a good place to illustrate the idea that different values can evaluate to `True` or `False` while testing conditions. Remember that strings resolve to `True` unless they're blank. Let's test this with an `if` statement:

```
>>> if '':
...     print('a blank string acts like True')
... else:
...     print('a blank string acts like false')
...
a blank string acts like false
```

If you'd used a string containing any characters at all, including a single space, then the preceding example would have resolved to `True` instead of `False`. If you have a Python console open, go ahead and try it and see for yourself. Let's look at one more example that resolves to `True` because the list isn't empty:

```
>>> if [1]:
...     print('a non-empty list acts like True')
... else:
```

```
...     print('a non-empty list acts like False')
...
a non-empty list acts like True
```

You can use this same idea to test that a number isn't equal to zero, because zero is the same as `False`, but any other number, positive or negative, will be treated as `True`.

2.5.2 While statements

A `while` statement executes a block of code as long as a condition is `True`. The condition is evaluated, and if it's `True`, then the code is executed. Then the condition is checked again, and if it's still `True`, then the code executes again. This continues until the condition is `False`. If the condition never becomes `False`, then the code will run forever, which is called an *infinite loop* and is a scenario you definitely want to avoid. Here's an example of a `while` loop:

```
>>> n = 0
>>> while n < 5:
...     print(n)
...     n += 1
...
0
1
2
3
4
```

The `+=` syntax means "increment the value on the left by the value on the right," so `n` is incremented by `1`. Once `n` is equal to `5`, it's no longer less than `5`, so the condition becomes `False` and the indented code isn't executed again.

2.5.3 For statements

A `for` statement allows you to iterate over a sequence of values and do something for each one. When you write a `for` statement, you not only provide the sequence to iterate over, but you also provide a variable name. Each time through the loop, this variable contains a different value from the sequence. This example iterates through a list of names and prints a message for each one:

```
>>> names = ['Chris', 'Janet', 'Tami']
>>> for name in names:
...     print('Hello {}!'.format(name))
...
Hello Chris!
Hello Janet!
Hello Tami!
```

The first time through the loop, the `name` variable is equal to `'Chris'`, the second time it holds `'Janet'`, and the last time it is equal to `'Tami'`. I called this variable `name`, but it can be called anything you want.

THE RANGE FUNCTION

The range function makes it easy to iterate over a sequence of numbers. Although this function has more parameters, the simplest way to use it is to provide a number *n*, and it will create a sequence from 0 to *n*-1. For example, this will count how many times the loop was executed:

```
>>> n = 0
>>> for i in range(20):
...     n += 1
...
>>> print(n)
20
```

The variable `i` wasn't used in this code, but nothing is stopping you from using it. Let's use it to calculate the factorial of 20, although this time we'll provide a starting value of `1` for the sequence, and have it go up to but not include the number `21`:

```
>>> n = 1
>>> for i in range(1, 21):
...     n = n * i
...
>>> print(n)
2432902008176640000
```

You'll see in later chapters that this variable is also useful for accessing individual items in a dataset when they aren't directly iterable.

2.5.4 break, continue, and else

A few statements apply to `while` and `for` loops. The first one, `break`, will kick execution completely out of the loop, as in this example that stops the loop when `i` is equal to `3`:

```
>>> for i in range(5):
...     if i == 3:
...         break
...     print(i)
...
0
1
2
```

Without the break `statement`, this loop would have printed the numbers 0 through 4.

The `continue` statement jumps back up to the top of the loop and starts the next iteration, skipping the rest of the code that would normally be executed during the current loop iteration. In this example, `continue` is used to skip the code that prints `i` if it's equal to `3`:

```
>>> for i in range(5):
...     if i == 3:
...         continue
...     print(i)
...
0
```

```
1
2
4
```

Loops can also have an `else` statement. Code inside of this clause is executed when the loop is done executing, unless the loop was stopped with `break`. Here we'll check to see if the number 2 is in a list of numbers. If it is, we'll break out of the loop. Otherwise, the `else` clause is used to notify us that the number wasn't found. In the first case, the number is found, `break` is used to exit the loop, and the `else` statement is ignored:

```
>>> for i in [0, 5, 7, 2, 3]:
...     if i == 2:
...         print('Found it!')
...         break
... else:
...     print('Could not find 2')
...
Found it!
```

But if the number isn't found, so `break` is never called, then the `else` clause is executed:

```
>>> for i in [0, 5, 7, 3]:
...     if i == 2:
...         print('Found it!')
...         break
... else:
...     print('Could not find 2')
...
Could not find 2
```

You could use this pattern to set a default value for something if an appropriate value wasn't found in a list. For example, say you needed to find and edit a file with a specific format in a folder. If you can't find a file with the correct format, you need to create one. You could loop through the files in the folder, and if you found an appropriate one you could break out of the loop. You could create a new file inside the `else` clause, and that code would only run if no suitable existing file had been found.

2.6 Functions

If you find that you reuse the same bits of code over and over, you can create your own function and call that instead of repeating the same code. This makes things much easier and also less error-prone, because you won't have nearly as many places to make typos. When you create a function, you need to give it a name and tell it what parameters the user needs to provide to use it. Let's create a simple function to calculate a factorial:

```
def factorial(n):
    answer = 1
    for i in range(1, n + 1):
        answer = answer * i
    return answer
```

The name of this function is `factorial`, and it takes one parameter, `n`. It uses the same algorithm you used earlier to calculate a factorial and then uses a `return` statement to send the answer back to the caller. You could use this function like this:

```
>>> fact5 = factorial(5)
```

Functions can also have optional parameters that the user doesn't need to provide. To create one of these, you must provide a default value for it when you create the function. For example, you could modify factorial to optionally print out the answer:

```
def factorial(n, print_it=False):
    answer = 1
    for i in range(1, n + 1):
        answer = answer * i
    if print_it:
        print('{0}! = {1}'.format(n, answer))
    return answer
```

If you were to call this function with only a number, nothing would get printed because the default value of `print_it` is `False`. But if you pass `True` as the second parameter, then a message will print before the answer is returned:

```
>>> fact5 = factorial(5, True)
5! = 120
```

It's easy to reuse your functions by saving them in a .py file and then importing them the way you would any other module. The one hitch is that your file needs to be in a location where Python can find it. One way to do this is to put it in the same folder as the script that you're running. For example, if the `factorial` function was saved in a file called myfuncs.py, you could import `myfuncs` (notice there's no .py extension) and then call the function inside of it:

```
import myfuncs
fact5 = myfuncs.factorial(5)
```

Because certain characters aren't allowed in module names, and module names are only filenames without the extension, you need to be careful when naming your files. For example, underscores are allowed in module names, but hyphens aren't.

2.7 Classes

As you work through this book, you'll come across variables that have other data and functions attached to them. These are objects created from classes. Although we won't cover how to create your own classes in this book, you need to be aware of them because you'll still use ones defined by someone else. Classes are an extremely powerful concept, but all you need to understand for the purposes of this book are that they're data types that can contain their own internal data and functions. An object or variable that is of this type contains these data and functions, and the functions operate on that particular object. You saw this with several of the data types we looked at

earlier, such as lists. You can have a variable of type `list`, and that variable contains all of the functions, such as `append`, that come with being a list. When you call `append` on a list, it only appends data to that particular list and not to any other list variables you might have.

Classes can also have methods that don't apply to a particular object, but to the data type itself. For example, the Python datetime module contains a class, or type, called `date`. Let's get that data type out of the module and then use it to create a new date object, which we can then ask which day of the week it is, where Monday is 0 and Sunday is 6:

```
>>> import datetime
>>> datetype = datetime.date
>>> mydate = datetype.today()
>>> mydate
datetime.date(2014, 5, 18)
>>> mydate.weekday()
6
```

The `datetype` variable holds a reference to the `date` type itself, not to a particular date object. The data type has a method, `today`, that creates a new date object. The date object stored in the `mydate` variable stores date information internally and uses that to determine what day of the week the date refers to, Sunday in this case. You couldn't ask the `datetype` variable what weekday it was, because it doesn't contain any information about a particular date. You don't need to get a reference to the data type and could have created `mydate` with `datetime.date.today()`. Now suppose you want to find out what day of the week May 18 was in 2010. You can create a new date object based on the existing one, but with the year changed, and then you can ask the new one what day of the week it represents:

```
>>> newdate = mydate.replace(year=2010)
>>> newdate
datetime.date(2010, 5, 18)
>>> newdate.weekday()
1
```

Apparently May 18, 2010, was a Tuesday. The original `mydate` variable hasn't changed, and will still report that it refers to a Sunday.

You'll use objects created from classes throughout this book. For example, whenever you open a dataset, you'll get an object that represents that dataset. Depending on the type of data, that object will have different information and functions associated with it. Obviously, you need to know about the classes being used to create these objects, so that you know what data and functions they contain. The GDAL modules contain fairly extensive classes, which are documented in appendixes B, C, and D. (Appendixes C through E are available online on the Manning Publications website at www.manning.com/books/geoprocessing-with-python.)

2.8 Summary

- The Python interpreter is useful for learning how things work or trying out small bits of code, but writing scripts is more efficient for running multiple lines of code. Plus, you can save scripts and use them later, which is one of the main reasons for programming.
- Modules are libraries of code that you can load into your script and use. If you need to do something with Python, chances are good that somewhere a module exists that will help you out, no matter what it is you're trying to do.
- Get used to storing data in variables, because it will make your code much easier to adapt later.
- Python has a few core data types, all of which are extremely useful for different types of data and different situations.
- You can use control flow statements to change which lines of code execute based on various conditions or to repeat the same code multiple times.
- Use functions to make your code reusable.