Supervised classification

Objects on the Earth reflect different wavelengths of light depending on the materials that they are made of. In remote sensing, analysts collect wavelength signatures for specific types of land cover (for example, concrete) and build a library for a specific area. A computer can then use this library to automatically locate classes in the library in a new image of the same area.

Unsupervised classification

In an unsupervised classification, a computer groups pixels with similar reflectance values in an image without any other reference information other than the histogram of the image.

Creating the simplest possible Python GIS

Now that we have a better understanding of geospatial analysis, the next step is to build a simple GIS using Python called `SimpleGIS`. This small program will be a technically complete GIS with a geographic data model and the ability to render the data as a visual thematic map showing the population of different cities.

The data model will also be structured so that you can perform basic queries. Our `SimpleGIS` will contain the state of Colorado, three cities, and population counts for each city.

Most importantly, we will demonstrate the power and simplicity of Python programming by building this tiny system in pure Python. We will only use modules available in the standard Python distribution without downloading any third-party libraries.

Getting started with Python

As stated earlier, this book assumes that you have some basic knowledge of Python. The examples in this book are based on Python 3.4.3, which you can download here:

`https://www.python.org/downloads/release/python-343/`

The only module used in the following example is the `turtle` module that provides a very simple graphics engine based on the `Tkinter` library included with Python. If you used the installers for Windows or Mac OS X, the `Tkinter` library should be included already. If you compiled Python yourself or are using a distribution from somewhere besides `Python.org`, then make sure that you can import the `turtle` module by typing the following at a command prompt to run the `turtle` demo script:

```
python -m turtle
```

installation not needed

If your Python distribution does not have Tkinter, you can find information on installing it from the following page. The information is for Python 2.3 but the process is still the same:

`http://tkinter.unpythonic.net/wiki/How_to_install_Tkinter`

The official Python wiki page for Tkinter can be found here:

`https://wiki.python.org/moin/TkInter`

The documentation for Tkinter is in the Python Standard Library documentation that can be found at `https://docs.python.org/2/library/tkinter.html`.

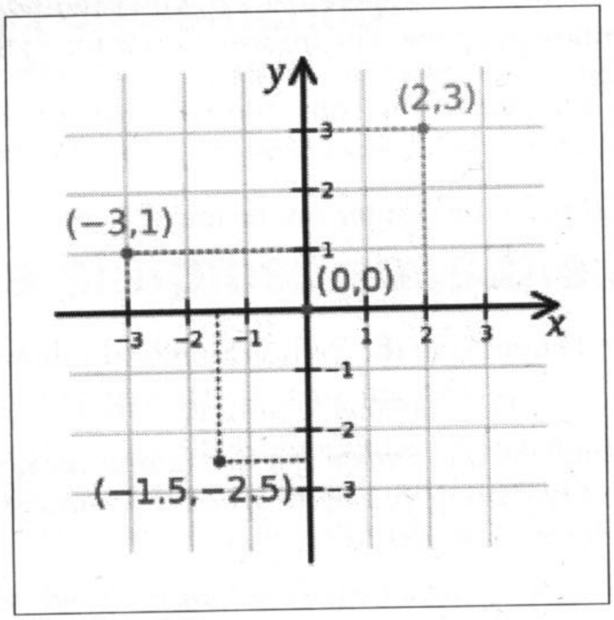
If you are new to Python, *Dive into Python* is a free online book, which covers all the basics of Python and will bring you up to speed. For more information, refer to `http://www.diveintopython.net`.

Building SimpleGIS

START HERE

The code is divided into two different sections. The first is the data model section and the second is the map renderer that draws this data. For the data model, we will use simple Python lists. A Python list is a native data type, which serves as a container for other Python objects in a specified order. Python lists can contain other lists and are great for simple data structures. They also map well to more complex structures or even databases if you decide you want to develop your script further.

The second portion of the code will render the map using the Python turtle graphics engine. We will have only one function in the GIS that converts the world coordinates, in this case, longitude and latitude, into pixel coordinates. All graphics engines have an origin point of (0,0) that is usually in the top-left or lower-left corner of the canvas. Turtle graphics are designed to teach programming visually. The turtle graphics canvas uses an origin of **(0,0)** in the center, similar to a graphing calculator. The following image illustrates the type of Cartesian graph that the turtle module uses. In the following graph, some points are plotted in both positive and negative space:



This also means that the turtle graphics engine can have negative pixel coordinates, which is uncommon for graphics canvases. However, for this example, the `turtle` module is the quickest and simplest way to render our map.

Step by step

You can run this program interactively in the Python interpreter or you can save the complete program as a script and run it. The Python interpreter is an incredibly powerful way to learn new concepts because it gives you real-time feedback on errors or unexpected program behavior. You can easily recover from these issues and try something else until you get the results that you want:

1. In Python, you usually import modules at the beginning of the script so we'll import the `turtle` module first. We'll use Python's import as feature to assign the module the name `t` to save space and time when typing turtle commands:

```
import turtle as t
```

2. Next, we'll set up the data model starting with some simple variables that allow us to access list indexes by name instead of numbers to make the code easier to follow. Python lists index the contained objects starting with the number `0`. So, if we want to access the first item in a list called `myList`, we would reference it as follows:

```
myList[0]
```

3. To make our code easier to read, we can also use a variable name assigned to commonly used indexes:

```
firstItem = 0
myList[firstItem]
```

Code until point 8 is provided in simpleGIS.zip

In computer science, assigning commonly used numbers to an easy-to-remember variable is a common practice. These variables are called constants.

So, for our example, we'll assign constants for some common elements used for all the cities. All cities will have a name, one or more points, and a population count:

```
NAME = 0
POINTS = 1
POP = 2
```

4. Now, we'll set up the data for Colorado as a list with name, polygon points, and population. Note that the coordinates are a list within a list:

```
state = ["COLORADO", [[-109, 37],[-109, 41],[-102, 41],[-102,
37]], 5187582]
```

5. The cities will be stored as nested lists. Each city's location consists of a single point as a longitude and latitude pair. These entries will complete our GIS data model. We'll start with an empty list called cities and then append the data to this list for each city:

```
cities = []
cities.append(["DENVER",[-104.98, 39.74], 634265])
cities.append(["BOULDER",[-105.27, 40.02], 98889])
cities.append(["DURANGO",[-107.88,37.28], 17069])
```

6. We will now render our GIS data as a map by first defining a map size. The width and height can be anything that you want depending on your screen resolution:

```
map_width = 400
map_height = 300
```

7. In order to scale the map to the graphics canvas, we must first determine the bounding box of the largest layer, which is the state. We'll set the map's bounding box to a global scale and reduce it to the size of the state. To do so, we'll loop through the longitude and latitude of each point and compare it with the current minimum and maximum *x* and *y* values. If it is larger than the current maximum or smaller than the current minimum, we'll make this value the new maximum or minimum, respectively:

```
minx = 180
maxx = -180
miny = 90
maxy = -90
forx,y in state[POINTS]:
if x < minx: minx = x
  elif x > maxx: maxx = x
  if y < miny: miny = y
  elif y > maxy: maxy = y
```

Code until point 8 is provided in simpleGIS.zip

8. The second step to scaling is to calculate a ratio between the actual state and the tiny canvas that we will render it on. This ratio is used for coordinate to pixel conversion. We get the size along the *x* and *y* axes of the state and then we divide the map width and height by these numbers to get our scaling ratio:

```
dist_x = maxx - minx
dist_y = maxy - miny
x_ratio = map_width / dist_x
y_ratio = map_height / dist_y
```

9. The following function, called `convert()`, is our only function in `SimpleGIS`. It transforms a point in the map coordinates from one of our data layers to pixel coordinates using the previous calculations. You'll notice that, at the end, we divide the map width and height in half and subtract it from the final conversion to account for the unusual center origin of the turtle graphics canvas. Every geospatial program has some form of this function:

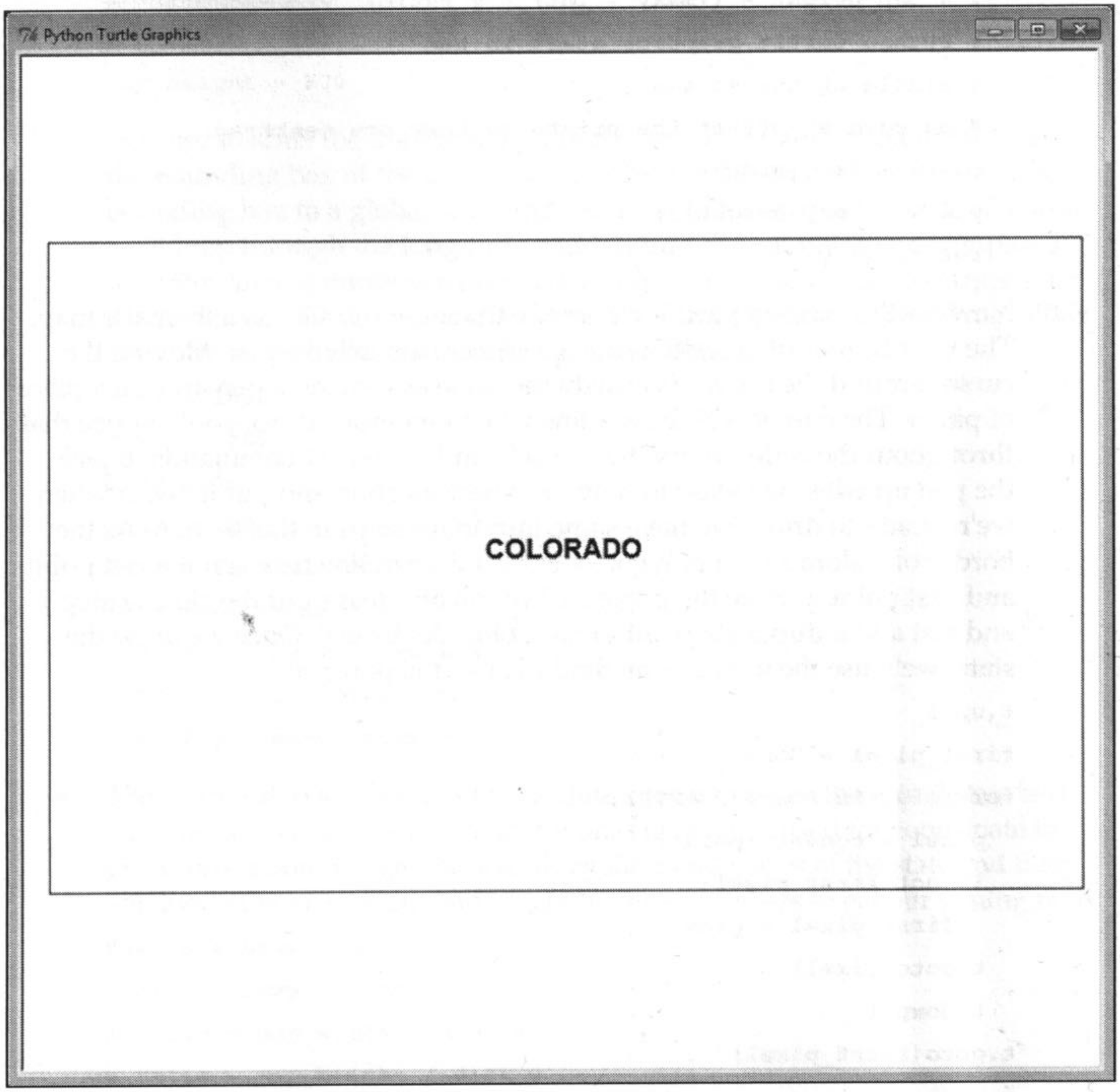
```
def convert(point):
  lon = point[0]
```

```
  lat = point[1]
  x = map_width - ((maxx - lon) * x_ratio)
  y = map_height - ((maxy - lat) * y_ratio)
  # Python turtle graphics start in the
  # middle of the screen
  # so we must offset the points so they are centered
  x = x - (map_width/2)
  y = y - (map_height/2)
  return [x,y]
```

10. Now for the exciting part! We're ready to render our GIS as a thematic map. The `turtle` module uses the concept of a cursor called a pen. Moving the cursor around the canvas is exactly the same as moving a pen around a piece of paper. The cursor will draw a line when you move it. So, you'll notice that throughout the code, we use the `t.up()` and `t.down()` commands to pick the pen up when we want to move to a new location and put it down when we're ready to draw. We have some important steps in this section. As the border of Colorado is a polygon, we must draw a line between the last point and first point to close the polygon. We can also leave out the closing step and just add a duplicate point to the Colorado dataset. Once we draw the state, we'll use the `write()` method to label the polygon:

```
t.up()
first_pixel = None
for point in state[POINTS]:
  pixel = convert(point)
  if not first_pixel:
    first_pixel = pixel
  t.goto(pixel)
  t.down()
t.goto(first_pixel)
t.up()
t.goto([0,0])
t.write(state[NAME], align="center", font=("Arial",16,"bold"))
```

11. If we were to run the code at this point, we would see a simplified map of the state of Colorado, as shown in the following screenshot:



If you do try to run the code, you'll need to temporarily add the following line at the end, or the Tkinter window will close as soon as it finishes drawing:

```
t.done()
```

12. Now, we'll render the cities as point locations and label them with their names and population. As the cities are a group of features in a list, we'll loop through them to render them. Instead of drawing lines by moving the pen around, we'll use the turtle `dot()` method to plot a small circle at the pixel coordinate returned by our `SimpleGISconvert()` function. We'll then label the dot with the city's name and add the population. You'll notice that we must convert the population number to a string in order to use it in the turtle `write()` method. To do so, we use Python's built-in function, `str()`:

```
#for city in cities:
pixel = convert(city[POINTS])
t.up()
t.goto(pixel)
# Place a point for the city
t.dot(10)
# Label the city
t.write(city[NAME] + ", Pop.: " + str(city[POP]), align="left")
t.up()
```

13. Now we will perform one last operation to prove that we have created a real GIS. We will perform an attribute query on our data to determine which city has the largest population. Then, we'll perform a spatial query to see which city lies the furthest west. Finally, we'll print the answers to our questions on our thematic map page safely out of the range of the map.
14. As our query engine, we'll use Python's built-in `min()` and `max()` functions. These functions take a list as an argument and return the minimum and maximum values of this list. These functions have a special feature called a key argument that allows you to sort complex objects. As we are dealing with nested lists in our data model, we'll take advantage of the key argument in these functions. The key argument accepts a function that temporarily alters the list for evaluation before a final value is returned. In this case, we want to isolate the population values for comparison and then the points. We could write a whole new function to return the specified value, but we can use Python's lambda keyword instead. The lambda keyword defines an anonymous function that is used inline. Other Python functions can be used inline, for example, the string function, `str()`, but they are not anonymous. This temporary function will isolate our value of interest.
15. So our first question is, which city has the largest population?

```
biggest_city = max(cities, key=lambda city:city[POP])
t.goto(0,-200)
t.write("The biggest city is: " +  biggest_city[NAME])
```

16. The next question is, which city lies the furthest west?

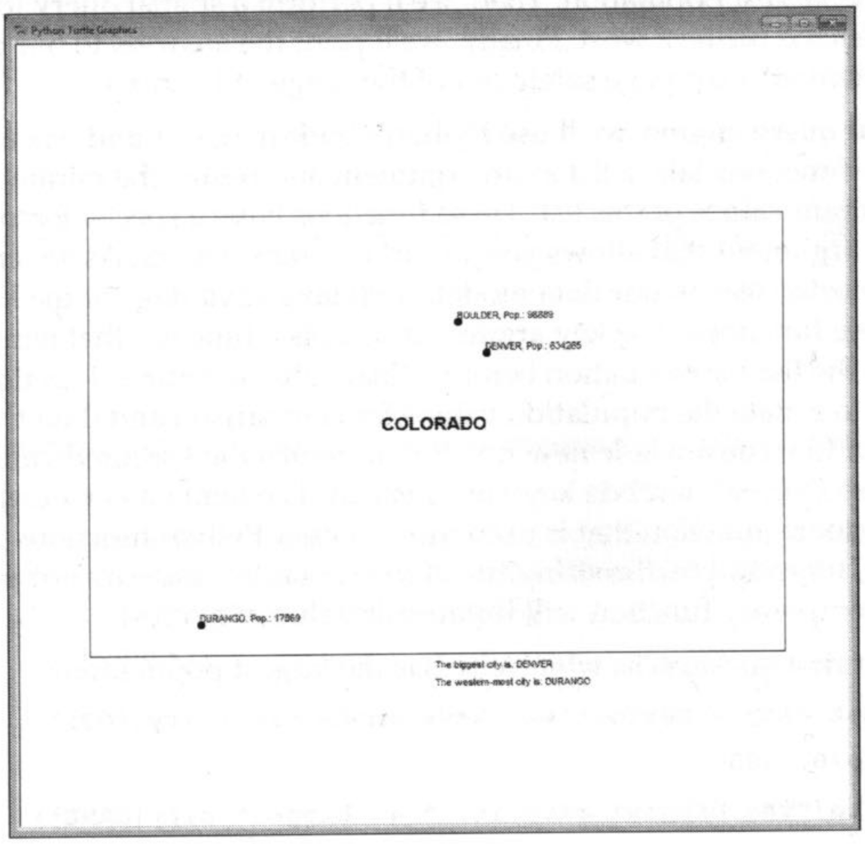
    ```
    western_city = min(cities, key=lambda city:city[POINTS])
    t.goto(0,-220)
    t.write("The western-most city is: " + western_city[NAME])
    ```

17. In the preceding query, we use Python's built in `min()` function to select the smallest longitude value and this works because we represented our city locations as longitude and latitude pairs. It is possible to use different representations for points including possible representations where this code would need modification to work correctly. However, for our `SimpleGIS`, we are using a common point representation to make it as intuitive as possible.

These last two commands are just for clean up purposes. First, we hide the cursor. Then, we call the turtle `done()` method, which will keep the turtle graphics window with our map open until we choose to close it using the close handle at the top of the window.

```
t.pen(shown=False)
t.done()
```

Whether you followed along using the Python interpreter or you ran the complete program as a script, you should see the following map rendered in real time:



Congratulations! You have followed in the footsteps of Paleolithic hunters, the *father of GIS* Dr. Roger Tomlinson, geospatial pioneer Howard Fisher, and game-changing humanitarian programmers to create a functional, extensible, and technically complete geographic information system. It took less than 60 lines of pure Python code! You will be hard-pressed to find a programming language that can create a complete GIS using only its core libraries in such a finite amount of readable code as Python. Even if you did, it is highly unlikely that the language would survive the geospatial Python journey that you'll take through the rest of this book.

As you can see, there is lots of room for expansion of `SimpleGIS`. Here are some other ways that you might expand this simple tool using the reference material for Tkinter and Python linked at the beginning of this section:

- Create an overview map in the top-right corner with a U.S. border outline and Colorado's location in the U.S.
- Add color for visual appeal and further clarity
- Create a map key for different features
- Make a list of states and cities and add more states and cities
- Add a title to the map
- Create a bar chart to compare population numbers visually

The possibilities are endless. `SimpleGIS` can also be used as a way to quickly test and visualize geospatial algorithms that you come across. If you want to add more data layers, you can create more lists, but these lists would become difficult to manage. In this case, you can use another Python module included in the standard distribution. The `SQLite` module provides a SQL-like database in Python that can be saved to disk or run in memory.

Summary

Well done! You are now a geospatial analyst. In this chapter, you learned the state of the art of geospatial analysis through the story of Ushahidi and the Ebola virus. You learned the history of geospatial analysis and the technologies that support it. You became familiar with foundational GIS and remote sensing concepts that will serve you through the rest of the book. You actually built a working GIS that can be expanded to do whatever you can imagine! In the next chapter, we'll tackle the data formats that you'll encounter as geospatial analysts. Geospatial analysts spend far more time dealing with data than actually performing analysis. Understanding the data that you're working with is essential to working efficiently and having fun.