9

Reading and writing raster data

This chapter covers

- Understanding raster data basics
- Introducing GDAL
- Reading and writing raster data
- Resampling data

If you have a geographic dataset that's made of continuous data such as elevation or temperature, it's probably a raster dataset. Spectral data such as aerial photographs and satellite imagery are also stored this way. These types of datasets don't assume strict boundaries exist between objects in the way that vector datasets do. Think of a digital photograph and how each pixel can be a slightly different color than the pixels next to it. The fact that pixel values can vary continuously like this makes for a much better-looking photo than if there were only a few colors to choose from. This trait also makes rasters appropriate for continuously varying data such as elevation.

Working with raster datasets is different from working with vectors. Instead of having individual geometries, you have a collection of pixels which is essentially a large two- or three-dimensional array of numbers. A raster dataset is made of bands instead of layers, and each of these bands is a two-dimensional array. The collection of bands becomes a 3D array. It's a different way of thinking about spatial data, and if you have a math phobia, this description might make it sound scary, too. But you'll soon see that it's not.

In this chapter you'll learn basic theory of raster data, including tips for keeping them to manageable sizes. Then you'll see how to use Python and GDAL to read these datasets into memory and how to write them back out to disk. The easiest case is to read and write an entire dataset at once, but sometimes you don't need the entire spatial extent, and other times the amount of memory is a limiting factor, so you'll also learn how to deal with a spatial subset of the data. It's also possible to change the pixel size while reading or writing, and you'll see how to do that as well.

9.1 Introduction to raster data

As mentioned, raster datasets can hold pretty much any type of data you'd like. That doesn't mean that it's always a good idea to use rasters, however. Objects that can be thought of as points, lines, or polygons are usually better left as vectors. For example, country boundaries lend themselves perfectly to a polygon vector dataset. This same data could be stored in a raster, but it would take up more disk space and the boundaries wouldn't be nice, smooth lines. You also couldn't use vector data analysis functions such as buffering and intersecting. These would still be possible using raster techniques, but you'd be better off sticking to vector in this case.

Raster is a perfect choice when values change continuously instead of at sharply defined boundaries. This includes common datasets such as elevation, slope, aspect, precipitation, temperature, and satellite data, but it can include many other things, too. It could be evapotranspiration, distance from roads, soil moisture, or anything else you might need to model as a continuous variable. Sometimes you need what would normally be vector data to be represented as a raster. For example, rivers and streams are good candidates for vector data, but rasters are required for modeling flow accumulation or groundwater flow, such as what would be needed to track the flow of a contaminant in the water supply.

Also, raster datasets don't have to contain continuous data. In fact, I see many rasters made of categorical data such as land cover type. One reason for this is that rasters are used in the models used to produce these datasets in the first place. For example, land cover models typically use visible and nonvisible wavelengths of light from satellite imagery, along with ancillary data such as elevation. The model output is a raster because the inputs are rasters, and it makes sense to leave it that way. It also makes the data easy to use as an input to other raster-based models.

Other examples include viewshed analysis, which takes topology into account when determining what's visible from a certain location. Maybe a ski resort would use this to decide where to locate a restaurant for the best views, and I know of a case where this type of analysis is being used to determine if ground bird nesting sites are visible to hawks perched on power lines (unpublished, but see Hovick et al.[1] for related research). Speaking of wildlife, rasters can also be used for habitat modeling, which might be done purely for the sake of knowledge, or to help select conservation areas. You might think of elevation as a fairly static dataset, but I know researchers who use ground-based lasers (in the form of LIDAR systems) to create elevation models of riverbeds, and then they do the same thing after a flood event so they can compare the before and after elevation profiles (Schaffrath et al.[2]). The possibilities are endless, really.

Now let's talk a bit about the details of raster datasets. You can probably envision a digital photograph as a two-dimensional array of pixels. In fact, that's what we talk about when discussing the dimensions of a photo—the numbers of rows and columns in that 2D array. This is what raster datasets are, except that they aren't limited to two dimensions. They can have a third dimension in the form of bands. Digital photos have multiple bands, too, although you don't usually think of them that way. But they have one each for red, green, and blue wavelengths of light. Your computer (or printer) combines these together to produce the colors you see on your monitor screen. You're familiar with this concept if you've ever created a webpage and specified a color using HEX notation, where the first two numbers correspond to red, the second two to green, and the last two to blue, or RGB notation where you provide a separate number for each of these colors.

Also, just as with a photograph, each band in a dataset has the same numbers of rows and columns, so the pixels from one fall in the same spatial location as the pixels in another. If the pixels for each individual color in a photograph didn't line up correctly, I imagine the results would look fairly blurry.

Obviously not all raster datasets are photographs, so pixel values don't have to correspond to colors. Pixel values in a digital elevation model (DEM), for example, correspond to elevation values. Generally DEM datasets only contain one band, because elevation is the only value required to create a useful dataset. Figure 9.1 shows a single-band raster landcover map for the state of Utah, where each unique pixel value represents a different landcover classification. This dataset contains discrete, rather than continuous, data.

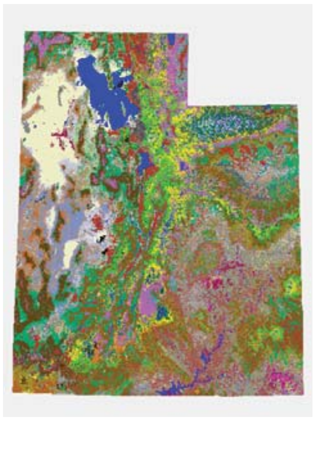

Figure 9.1 A landcover classification map, where each unique pixel value corresponds to a specific landcover classification

[1] Hovick, T. J., Elmore, R. D., Dahlgren, D. K., Fuhlendorf, S. D., Engle, D. M. 2014. REVIEW: Evidence of negative effects of anthropogenic structures on wildlife: a review of grouse survival and behavior. Journal of Applied Ecology, 51: 1680–1689. DOI: 10.1111/1365-2664.12331.

[2] Schaffrath, K. R., P. Belmont, and J.M Wheaton. 2015. Landscape-scale geomorphic change detection: Quantifying spatially variable uncertainty and circumventing legacy data issues. Geomorphology, 250: 334-348. DOI: 10.1016/j.geomorph.2015.09.020.

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.

Satellite imagery, on the other hand, contains measurements of various wavelengths, many of which aren't visible to our eyes. While the image might contain bands corresponding to visible red, green, and blue wavelengths, there might also be bands for infrared or thermal radiation. False color images such as those shown in figure 9.2 are created by displaying an infrared band along with visible light. This figure illustrates another use case for satellite imagery in the form of raster data. The image on the left was created using visible light wavelengths, like a traditional photograph. The camera also captured a near infrared wavelength as another band at the same time, and it was used along with the visible red and green bands to create the false color image on the right. The near infrared band is brighter for growing vegetation, and it's displayed as red, so red areas are vegetation. This band combination is useful for monitoring vegetation. The field in the stadium and the practice fields outside are both green and look like grass in the natural color image, but the field inside the stadium is dark gray in the false color image, so it must be artificial. The practice fields outside the stadium are bright red, however, so they must be grass.

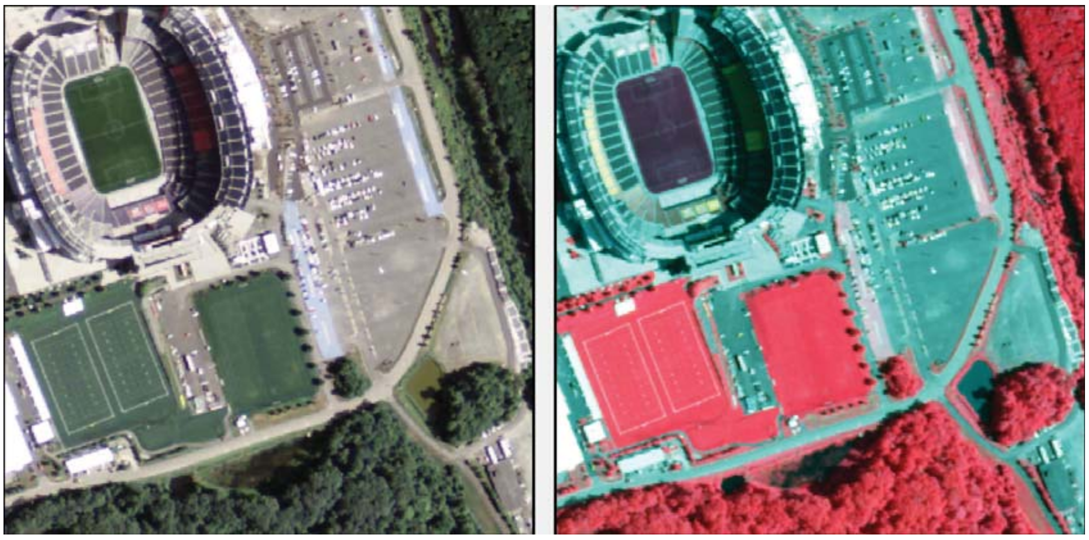

Figure 9.2 Two images of Gillette Stadium, home of the New England Patriots, in Foxborough, Massachusetts. The field inside the stadium looks like grass in the natural color image on the left, and it looks gray in the false color image on the right, which makes it clear that it's actually artificial. The practice fields, on the other hand, also look like grass in the natural color image on the left, but are red in the image on the right, signifying that they are indeed grass.

Let’s make another comparison to photographs. Although pictures you take with your phone might be geotagged, meaning that metadata in the image specifies where you were standing when you took the photo, each pixel doesn’t correspond to a specific location on the ground. That wouldn’t even make sense for most of the photos you take, but what if you took one from an airplane, looking straight down? A photo like that could be overlaid on a map if you had the appropriate spatial information. Having only the geotagged coordinates isn’t enough, even if you knew exactly what part of the photo the coordinates corresponded to, such as a corner or the center. For example, think how different your photo would look if you were in a Cessna flying relatively close to the ground as opposed to a 747 much higher up. Your photo would cover a much larger area in the second case, even if the photo was taken with the same camera and had the same number of rows and columns. The difference is the pixel size or the area on the ground that a single pixel covers. The pixels in the photo taken from the Cessna would each cover a smaller area than a pixel in the 747 photo. If you knew how much area they each covered, along with the coordinates of one of them, you could figure out how to stretch the photo so that it overlaid on a map. This is assuming that your camera was pointed exactly straight down so the pixels aren’t skewed to one side. It also assumes that you had things aligned perfectly so that the top of the photo was exactly north, although you could rotate the image to compensate for that.

Knowing the pixel size is important if you want to overlay your photo on a map, but you obviously need coordinates to go with it. With vector data it’s enough to know the spatial reference system because the coordinates for each feature are stored in the vertices. Given the SRS, each vertex can be placed in the correct location with lines drawn between them, and you have your geometry. Raster datasets don’t use vertices, and instead commonly use one set of coordinates, the pixel size, and the amount the dataset is rotated to determine coordinates for the rest of the image. This is called an *affine transformation*, and is a common way to georeference a raster dataset, although it isn’t the only way. The set of coordinates is generally for the upper-left corner of the image and is called the *origin*. For the simple (and common) case of a raster that has the top of the dataset facing north, you only need these coordinates and the pixel size to find the coordinates of any pixel in the image. All you need to do is figure out the offsets from the origin, multiply those by the pixel size, and add that to the origin coordinates. Figure 9.3 shows how to get the upper-left coordinates for the pixel in the fifth column and fourth row. The first row and column have offsets 0, so

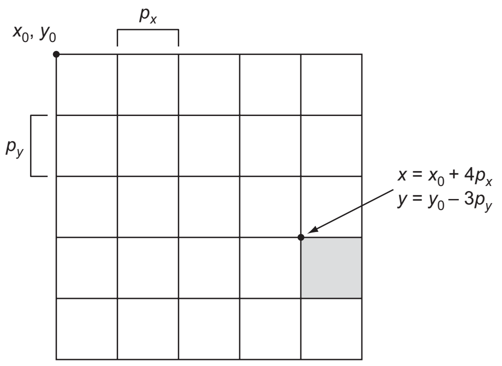

Figure 9.3 An example showing how to get the upper-left coordinates for the pixel in the fifth column and fourth row

your offsets in this case are 4 and 3. To get the easting coordinate, multiply the pixel width by 4 to get the distance across those four columns. Then add that to the origin's easting coordinate, and you're done. You can get northings the same way, but by using the row offset instead of the column. As you can see, it's extremely important that the origin coordinates are correct and using the right SRS, or else you can't calculate coordinates for any part of the dataset.

As you've no doubt noticed, the more pixels contained in a photo, the more disk space that photo requires. Similarly, raster datasets can take up a lot of space on disk and in RAM, so you should ensure that you're not using smaller pixels or larger data types than are necessary. For example, if your data are only good down to 10 meters, it doesn't make sense to have 1-meter pixels because the smaller pixels inside your 10-meter block will all have the same value. As a comparison, you may have seen compact digital cameras with more megapixels than high-end digital SLRs. The SLR still takes better photos, though, because higher quality data is collected. More pixels aren't a substitute for quality data or effective resolution. Not only would they fail to improve your data, all of these extra pixels would greatly increase the size of your file. Doubling the number of rows and columns doesn't double the size of the image. Instead, it quadruples it! For example, an image with 250 rows and 250 columns would have 250 x 250 = 62,500 pixels, while an image with 500 rows and columns has 250,000 pixels.

The data type you choose for your data is also important when it comes to storage space. For example, if your pixel values all fall in the range of 0 to 254, then you should use a byte data type (254 is the largest value a byte can hold). In this case, each pixel will take up a byte, or 8 bits, of disk space, no matter the value, unless you're using compression. If you were to store this same data as 32-bit integer, each pixel would take up four times as much space as before. You'd be taking up four times as much memory with absolutely no benefit. With a small dataset, this might not matter so much, but it certainly does with large datasets.

If rasters can be so large and take a while to process, how can they be drawn on your screen in a reasonable amount of time? This is where *overview layers* come in. You might have also heard them called *pyramid layers* or *reduced resolution datasets* (hence the .rrd extension that several types of overviews have). Overview layers are reduced resolution layers—they're rasters that cover the same area as the original, but are resampled to larger pixel sizes. A raster dataset can have many different overviews, each with a different resolution. When you're zoomed out and looking at the whole image, the coarse resolution layer is drawn. Because the pixels are so large, that layer doesn't take much memory and can draw quickly, but you can't tell the difference at that zoom level. As you zoom in, a higher resolution layer is drawn, but only the part you're viewing needs to be loaded and shown. If you zoom in enough, you'll see the original pixels, but because you're only looking at a small subset of the image, it still draws quickly.

Figure 9.4 shows how this works. Each successive overview layer has a pixel size twice as large as the previous one. All resolutions look the same when viewing the entire image, but you can see the difference when you're zoomed into a smaller area. The

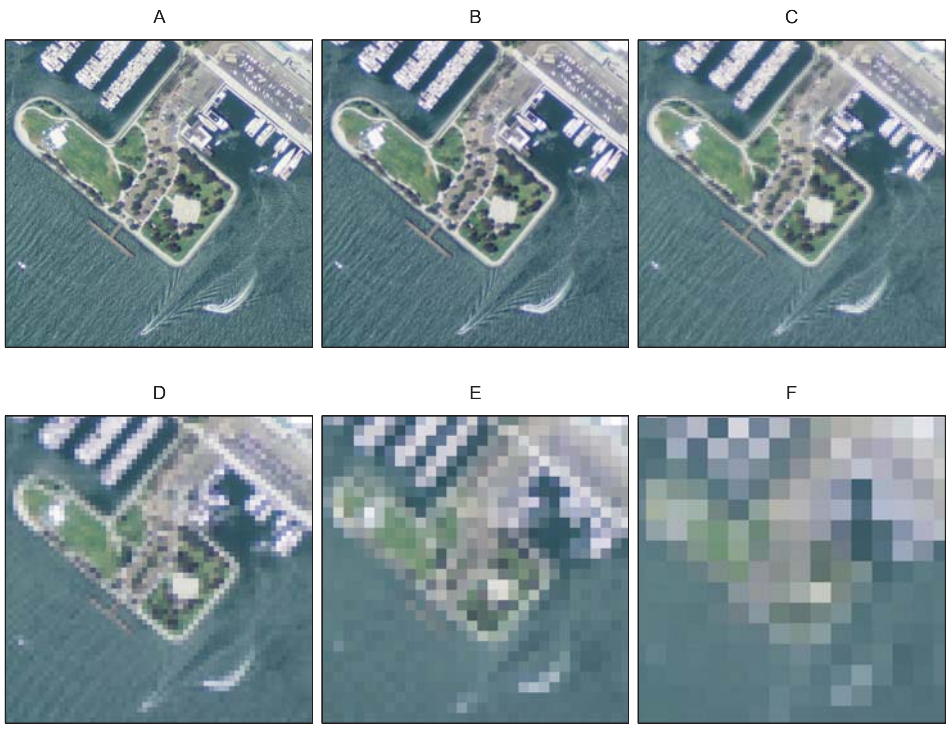

Figure 9.4 Example of how each overview is a coarser resolution than the previous one. The image in the upper left (A) is full resolution, and each successive image uses a pixel size twice as large as the previous one. The last image (F) only looks good when zoomed way out, but it draws much faster than the full image at that scale.

upper-left image in the figure is a full resolution (1 x 1-meter pixel) image of San Diego Harbor. You can see individual cars, boats, and trees. The middle-top image is the first set of overviews, with pixels 2 meters on a side. The pixels in the upper-right image are 4 x 4 meters. Along the bottom row, the pixels are 8, 16, and 32 meters on a side. If you're zoomed out so you can see all of San Diego, you can't tell the difference between the first and the last image, but the last one is a small fraction of the size of the full-resolution one and draws much faster. A good rule of thumb is to create overview layers of decreasing resolution until the coarsest one has 256 pixels or less in one dimension.

Incidentally, the ubiquitous web-mapping services use this same technique to display aerial photography, except that the reduced resolution layers are stored as collections of individual tiles. Your browser downloads whichever tiles are required to cover the area you're looking at, and as you zoom in, you get tiles that have a higher resolution and cover a smaller area, until eventually the resolution is so good that you can see your house or car.

Another aspect of raster datasets that influences access speed is how they're stored on disk. Rasters are made up of blocks, which have to do with how the data are arranged on the disk. As you'd expect, each format does this differently. (If not, they wouldn't really be different formats, would they?) Blocks of pixels are all physically stored next to each other on disk, so they can be accessed together efficiently. It's possible that other blocks belonging to the same image are stored on another part of the disk; this is the sort of problem you solve by defragmenting your drive. It's faster to grab data that are close to each other physically, just as it's faster for you to pull two books off the same shelf instead of from two different bookshelves. If you need to read or write data, it's most efficient to use blocks. For example, GeoTIFFs come in tiled and untiled formats. Untiled GeoTIFFs store each row of pixels as a block, but tiled ones use square sets of pixels instead, with 256 x 256 being a common block (or tile) size. It's faster to read data from a tiled GeoTIFF in the square chunks corresponding to blocks, but it's faster to deal with entire rows when it comes to untiled GeoTIFFs.

You're probably also wondering about data compression, because this is regularly used with digital photos in .jpeg format and can significantly reduce the size of a file. This is certainly possible, and multiple types of compression are available, depending on the data format. You might have heard of *lossy* versus *lossless* compression. Lossy compression loses information in the act of compressing the data. When saving .jpegs, you've probably noticed the compression quality option. The higher the quality, the less data you lose and the better the resulting image looks. The .png format, however, is lossless, which is why you aren't asked for a compression quality when saving one of those. That doesn't mean that the data can't be compressed. It means that you won't lose any data in the act of compression, and the image can be perfectly reconstructed into the original uncompressed dataset. If you plan to compress your data but you also need to perform analyses on it, make sure you select a lossless algorithm. Otherwise, your analysis won't be operating on the actual pixel values because several will have been lost. GeoTIFF is a popular lossless format.

I have one more important concept you need to understand if you're going to use raster data, and that's the difference between resampling methods. You don't have a one-to-one mapping between pixels when a raster is resampled to a different cell size or reprojected to another spatial reference system, so new pixel values have to be calculated. The simplest and fastest method, called *nearest-neighbor*, is to use the value from the old pixel that's closest to the new one. Another possible algorithm, shown in figure 9.5, is to take the average of the four closest pixels. Several other methods use

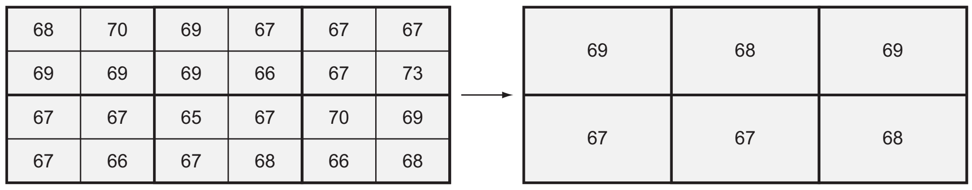

Figure 9.5 Simple resampling example, where the average of four pixels is used to calculate the value for a new pixel that covers the same extent as the four smaller pixels

multiple input pixels, such as bilinear, which uses a weighted average of the four closest input pixels.

You should always use nearest-neighbor resampling when dealing with datasets containing discrete values, such as the landcover classifications from figure 9.1. Otherwise, you might end up with a value that doesn't correspond to a classification, or with a number that denotes a completely unrelated classification. Continuous data, on the other hand, are well-suited to the other resampling methods. For example, taking an average elevation value makes perfect sense, and you'll get smoother output that way than if you used nearest-neighbor.

9.2 Introduction to GDAL

Now that the theory is out of the way, let's learn how to work with these datasets using GDAL. Numerous different file formats exist for raster data, and GDAL is an extremely popular and robust library for reading and writing many of them. The GDAL library is open source, but has a permissive license, so even many commercial software packages use it. Unfortunately, they don't necessarily use it to read as many formats as possible, so I have students and colleagues ask me on a regular basis if I can convert their data into a format their software can read. Every time I get asked this, I point the person to GDAL and its command-line utilities. They're usually amazed with what they can do with free software, and if they knew how to write their own code, they could do even more.

The GDAL library is well known for its ability to read and write so many different formats, but it also contains a few data processing functions such as proximity analysis. You'll still have to write your own processing code in many cases, but this is relatively easy for many types of analyses. There's a Python module called NumPy that's designed for processing large arrays of data, and you can use GDAL to read data directly into NumPy arrays. After manipulating the data however you need, using NumPy or another module that works with these arrays, you can write the array back out to disk as a raster dataset. It's a pretty painless process. You'll learn how to work directly with NumPy arrays in chapter 11.

I only use a small handful of raster formats on a regular basis, and I imagine that most readers of this book do the same, but nonetheless there are well over 100 different format drivers available for GDAL, listed online at http://www.gdal.org/formats_list.html. Each one of these drivers handles reading and writing a specific data format. You probably won't have all of them available with your version of GDAL, but they do exist. If you need a particular driver and can't find a precompiled GDAL binary with that specific one, you can always compile your own customized version of GDAL (although this might be tricky, especially if you have no experience with that sort of thing). Not all drivers support the same operations, however. While many support reading and writing, some are read-only, and others won't let you modify existing datasets, although you can create new ones. Assuming the driver supports the operation you have in mind, you use all of the drivers the same way. Most of my examples will use GeoTIFFs, but they'd work fine with other formats as well.

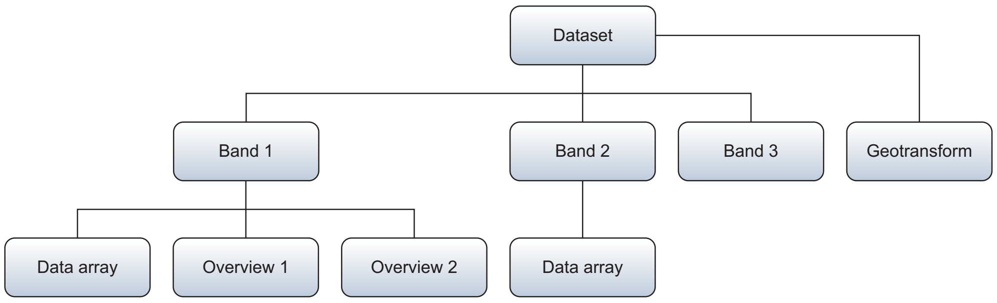

Figure 9.6 The basic structure of a GDAL dataset. Each dataset contains one or more bands, which in turn contain the pixel data.

The basic structure of a GDAL dataset is shown in figure 9.6 and matches what you've learned about raster datasets in general. Each dataset contains one or more bands, which in turn contain the pixel data and possibly overviews. The georeferencing information is contained in the dataset because all of the bands use the same information for this.

To illustrate how to use GDAL to read and write raster data, let's start with an example that combines three individual Landsat bands into one stacked image, as shown in figure 9.7. The Landsat program is a joint initiative between the United

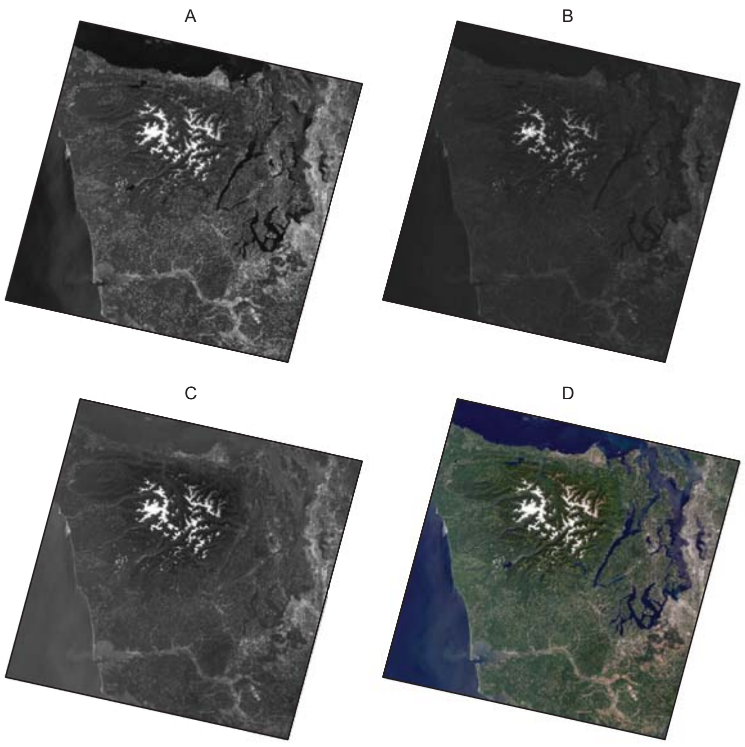

Figure 9.7 Red (A), green (B), and blue (C) Landsat bands are shown in black and white, and you can see that they look a bit different from each other. Part D shows these three bands stacked into an RGB image like that created by listing 9.1.

States Geological Survey (USGS) and the National Aeronautics and Space Administration (NASA), and has been collecting moderate-resolution satellite imagery worldwide since 1972. Landsat images are distributed by the USGS as a collection of GeoTIFFs, one for each collected band. With the exception of bands 6 (thermal) and 8 (panchromatic), each of these has a 30-meter resolution and because they're from the same Landsat scene, the same dimensions. This makes things easy, because the bands go directly on top of one another with no fiddling required. Listing 9.1 shows how to create a three-band dataset with these same dimensions, and then copy bands 3, 2, and 1 into it. These three bands correspond to red, green, and blue wavelengths of visible light, respectively, so putting them in this order will result in an RGB (red, green, blue) image that will look much like it would to your own eyes. Figure 9.7 shows the individual red, blue, and green bands in black and white, along with the resulting three-band natural color image. If you preview your images in a GIS, they'll probably look similar to this because the GIS will most likely stretch them. If you view them somewhere else, they'll look washed out compared to this. I chose to include the stretched images because you can't see any detail at all in the others on the printed page. Let's take a look at the following code.

Listing 9.1 Stacking individual raster bands into one image

```
import os
from osgeo import gdal                                      # Import GDAL

os.chdir(r'D:\osgeopy-data\Landsat\Washington')
band1_fn = 'p047r027_7t20000730_z10_nn10.tif'
band2_fn = 'p047r027_7t20000730_z10_nn20.tif'
band3_fn = 'p047r027_7t20000730_z10_nn30.tif'

in_ds = gdal.Open(band1_fn)                                 # Open the band 1
in_band = in_ds.GetRasterBand(1)                            # GeoTIFF

gtiff_driver = gdal.GetDriverByName('GTiff')                # Create a three-band
out_ds = gtiff_driver.Create('nat_color.tif',               # GeoTIFF with same
    in_band.XSize, in_band.YSize, 3, in_band.DataType)      # properties as band 1
out_ds.SetProjection(in_ds.GetProjection())
out_ds.SetGeoTransform(in_ds.GetGeoTransform())

in_data = in_band.ReadAsArray()                             # Copy pixel data from input
out_band = out_ds.GetRasterBand(3)                          # band to band 3 of output
out_band.WriteArray(in_data)

in_ds = gdal.Open(band2_fn)                                 # Copy pixel data from a
out_band = out_ds.GetRasterBand(2)                          # dataset instead of a band
out_band.WriteArray(in_ds.ReadAsArray())

out_ds.GetRasterBand(1).WriteArray(
    gdal.Open(band3_fn).ReadAsArray())

out_ds.FlushCache()                                         # Compute statistics on
for i in range(1, 4):                                       # each output band
    out_ds.GetRasterBand(i).ComputeStatistics(False)
out_ds.BuildOverviews('average', [2, 4, 8, 16, 32])    ◄─ Build over
                                                          views/pyramid layers
del out_ds
```

What happens here? Well, the first thing you do is import the `gdal` module. Then you set your current directory and specify which file corresponds to which Landsat band. Then you open the GeoTIFF containing the first band by passing the filename to `gdal.Open`. You also grab a handle to the first and only band inside the dataset, although you haven't read any data in yet. Notice that you use an index of 1 instead of 0 to get the first band. Band numbers always start with 1 when you use `GetRasterBand`, although I frequently forget and use 0 and then have to fix my error. Anyway, you need this band object before creating the output image because it has information you need.

TIP Remember that band indices start at 1 instead of 0.

Next you create a new dataset to copy the pixel data into. You have to use a driver object to create a new dataset, so you find the GeoTIFF driver and then use its `Create` function. Here's the full signature for that function:

```
driver.Create(filename, xsize, ysize, [bands], [data_type], [options])
```

- `filename` is the path to the dataset to create.
- `xsize` is the number of columns in the new dataset.
- `ysize` is the number of rows in the new dataset.
- `bands` is the number of bands in the new dataset. The default value is 1.
- `data_type` is the type of data that will be stored in the new dataset. The default value is `GDT_Byte`.
- `options` is a list of creation option strings. The possible values depend on the type of dataset being created.

Because you use the GeoTIFF driver, the output file will be a GeoTIFF no matter what file extension you give it. The extension isn't added automatically, however, so you do need to provide it. In this case, you call it nat_color.tif and save it in the D:\osgeopy-data\Landsat\Washington folder, because that's the current folder set with `os.chdir`. You're also required to provide the numbers of columns and rows when creating a new dataset, so you use the `XSize` and `YSize` properties, respectively, to get that information from the input band. The next argument to `Open` is the number of bands, and you want this new raster to have three of them. The next optional parameter is the data type, which has to be one of the values from table 9.1. You obtain this information from the input band, although you could have ignored it in this case because these images use the default type of `GDT_Byte`. You can also provide format-specific creation options, but you don't do that here. Because every format has its own options, you need to consult www.gdal.org/formats_list.html for your format of interest.

Table 9.1 GDAL data type constants

Constant	Data type
`GDT_Unknown`	Unknown
`GDT_Byte`	Unsigned 8-bit integer (byte)
`GDT_UInt16`	Unsigned 16-bit integer
`GDT_Int16`	Signed 16-bit integer
`GDT_UInt32`	Unsigned 32-bit integer
`GDT_Int32`	Signed 32-bit integer
`GDT_Float32`	32-bit floating point
`GDT_Float64`	64-bit floating point
`GDT_CInt16`	16-bit complex integer
`GDT_CInt32`	32-bit complex integer
`GDT_CFloat32`	32-bit complex floating point
`GDT_CFloat64`	64-bit complex floating point
`GDT_TypeCount`	Number of available data types

At this point you have an empty three-band dataset, but you probably want it to know what SRS it uses and where it's located on the planet. The next two lines take care of those details, and they're repeated here:

```
out_ds.SetProjection(in_ds.GetProjection())
out_ds.SetGeoTransform(in_ds.GetGeoTransform())
```

You get the projection (SRS) from the input dataset and copy it to the new dataset, and then you do the same for the *geotransform*. The geotransform is important because it provides the origin coordinates and pixel sizes, along with rotation values if the image isn't situated so the top faces north. As you learned earlier, the origin and pixel size are extremely important when it comes to placing the dataset in the correct spatial location. Although you don't have to add the projection and geotransform information before adding pixel values, I prefer to get this out of the way as soon as I create the new dataset.

After setting up your dataset, it's time to add pixel values. Because you already have the band object from the GeoTIFF for Landsat band 1, you can read the pixel values from it into a NumPy array. If you don't provide any parameters to `ReadAsArray`, then all pixel values are returned in a two-dimensional array with the same dimensions as the raster itself. At this point your `in_data` variable holds a two-dimensional array of pixel values:

```
in_data = in_band.ReadAsArray()
```

Now, because band 1 of a Landsat image is the blue band, you need to put that into the third band of your output image to get the bands in RGB order. The next thing you do is get the third band from `out_ds` and then use `WriteArray` to copy the values in the `in_data` array into the third band of your new dataset:

```
out_band = out_ds.GetRasterBand(3)
out_band.WriteArray(in_data)
```

You still need to add the green and red Landsat bands to your dataset, so then you open the second band's GeoTIFF. Notice that you don't get the band object from the dataset, though, because you're going to read pixel data directly from the dataset itself this time. Because the second Landsat band is the green one, you then get a handle to the second (green) band in your stacked dataset, and copy the data from the Landsat file to your stacked dataset:

```
in_ds = gdal.Open(band2_fn)
out_band = out_ds.GetRasterBand(2)
out_band.WriteArray(in_ds.ReadAsArray())
```

When you call `ReadAsArray` on a dataset, you get a three-dimensional array if the dataset you're reading from has multiple bands. Because the Landsat file only has one band, `ReadAsArray` on the dataset returns the same two-dimensional array that you'd get from the band object. Instead of saving the data into an intermediate variable, this time you immediately send it to the output band. Then you do the same thing for the red pixel values, but compress it into even less code. The result is the same, however:

```
out_ds.GetRasterBand(1).WriteArray(gdal.Open(band3_fn).ReadAsArray())
```

In the next bit of code, you compute statistics on each band in your dataset. This isn't strictly necessary, but it makes it easier for some software to display it nicely. The statistics include mean, minimum, maximum, and standard deviation. A GIS can use this information to stretch the data on the screen and make it look better. You'll see an example of how to stretch data manually in a later chapter. Before computing statistics, you have to ensure that the data have been written to disk instead of only cached in memory, so that's what the call to `FlushCache` does. Then you loop through the bands and compute the statistics for each one. Passing `False` to this function tells it that you want actual statistics instead of estimates, which it might get from overview layers (which don't exist yet) or from sampling a subset of the pixels. If an estimate is acceptable, then you can pass `True` instead; this will also make the calculation go faster because not every pixel needs to be inspected:

```
out_ds.FlushCache()
for i in range(1, 4):
    out_ds.GetRasterBand(i).ComputeStatistics(False)
```

The last thing you do is build overview layers for the dataset. Because these pixel values are continuous data, you use average interpolation instead of the default of nearest-neighbor. You also specify five levels of overviews to build. It happens that five levels are what you'd need to get tiles of size 256 for this particular image:

```
out_ds.BuildOverviews('average', [2, 4, 8, 16, 32])
```

Oh, and don't forget to delete the output dataset. This will happen automatically when the variable goes out of scope, but this may not be when your script finishes running if you're using an interactive Python environment. This is a regular occurrence when my students are working on their homework. They don't flush the cache or delete the variable, and their IDE doesn't release the dataset object when the script finishes, so they end up with an empty image and don't know why.

Other modules for working with raster data

If you'd like to play with a module that uses more "Pythonic" syntax but still harnesses the power of GDAL, check out rasterio at https://github.com/mapbox/rasterio. This module depends on GDAL and uses it internally to read and write data, but it tries to make the process of working with raster data a little easier.

Another module that might be of interest is imageio. This one is written in pure Python and doesn't rely on GDAL. It doesn't focus on geospatial data, but it can read and write many different raster formats, including video formats. You can read more about it at http://imageio.github.io/.

9.3 Reading partial datasets

In listing 9.1 you read and wrote entire bands of data at a time. You can break it up into chunks if you need to, however. This might be because you only need a spatial subset of the data to begin with, or maybe you don't have enough RAM to hold it all at once. Let's a take a look at how you can access subsets instead of the entire images.

The `ReadAsArray` function has several optional parameters, although they differ depending on whether you're using a dataset or a band.

Here's the signature for the band version:

```
band.ReadAsArray([xoff], [yoff], [win_xsize], [win_ysize], [buf_xsize],
                 [buf_ysize], [buf_obj])
```

- `xoff` is the column to start reading at. The default value is 0.
- `yoff` is the row to start reading at. The default value is 0.
- `win_xsize` is the number of columns to read. The default is to read them all.
- `win_ysize` is the number of rows to read. The default is to read them all.
- `buf_xsize` is the number of columns in the output array. The default is to use the `win_xsize` value. Data will be resampled if this value is different than `win_xsize`.

- `buf_ysize` is the number of rows in the output array. The default is to use the `win_ysize` value. Data will be resampled if this value is different than `win_ysize`.
- `buf_obj` is a NumPy array to put the data into instead of creating a new array. Data will be resampled, if needed, to fit into this array. Values will also be converted to the data type of this array.

The `xoff` and `yoff` parameters specify the column and row offsets, respectively, to start reading at. The default is to start reading at the first row and column. The `win_xsize` and `win_ysize` parameters indicate how many rows and columns to read, and the default is to read them all. The `buf_xsize` and `buf_ysize` parameters allow you to specify the size of the output array. If these values are different than the `win_xsize` and `win_ysize` values, then the data will be resampled as it's read to match the output array size. The `buf_obj` parameter is a NumPy array that the data will be stored in instead of a new array being created. The pixel data type will be changed to match the data type of this array. You'll get an error if you provide `buf_xsize` and `buf_ysize` values that don't match the dimensions of this array, but there's no reason to provide sizes in that case anyway, because they can be determined from the array itself.

For example, to read the three rows and six columns starting at row 6000 and column 1400 shown in figure 9.8, you could do something like the following:

```
data = band.ReadAsArray(1400, 6000, 6, 3)
```

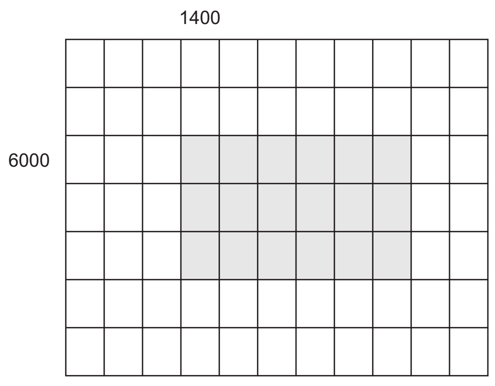

Figure 9.8
Use `ReadAsArray(1400, 6000, 6, 3)` to read three rows and six columns starting at row 6000 and column 1400.

If you need the pixel values as floating-point instead of byte, you can convert them using NumPy after you've read them in, like this:

```
data = band.ReadAsArray(1400, 6000, 6, 3).astype(float)
```

Or you could have GDAL do the conversion for you as it reads the data. To use this method, you create a floating-point array and then pass it as the `buf_obj` parameter to `ReadAsArray`. Make sure you create the array with the same dimensions as the data being read.

```
import numpy as np
data = np.empty((3, 6), dtype=float)
band.ReadAsArray(1400, 6000, 6, 3, buf_obj=data)
```

The NumPy `empty` function creates an array that hasn't been initialized with any values, so it contains garbage until you fill it somehow. The first parameter to the function is a tuple containing the dimensions of the array to create. If it's a two-dimensional array, the tuple contains the number of rows and then the number of columns. The `dtype` parameter is optional and specifies the type of data that the array will hold. If not provided, the array will hold floating-point numbers.

To write a data array out to a specific location in other dataset, pass the offsets to `WriteArray`. It will write out all data in the array you pass to the function, beginning at the offsets you provide.

```
band2.WriteArray(data, 1400, 6000)
```

One important thing to remember when reading partial datasets is that you have to make sure you don't try to read more data than exists, or you'll get an error. For example, if an image has 100 rows, and you ask it to start reading at offset 75 and read in 30 rows, that would go past the end of the image and will fail. A similar problem will occur if you pass an array to `WriteArray` that's too large to fit in the raster, given your starting offsets.

Access window out of range error messages

The following message means that I tried to read a 30 x 30 array from band 1 of testio.tif, starting at column 0 and row 75. The problem is that testio.tif only has 100 rows and 100 columns, so there aren't 30 rows to read if I start at number 75.

"ERROR 5: testio.tif, band 1: Access window out of range in RasterIO(). Requested (0,75) of size 30x30 on raster of 100x100."

How might you use this information to process a large dataset that won't fit in RAM? Well, one way would be to deal with a single block at a time. Remember that rasters store their data on disk in blocks. Because the data in a block are stored together on disk, it's efficient to process images in these chunks.

The basic idea is shown in figure 9.9. You start with the first block of rows and columns, and then go to the next block in either the x or y direction (this example uses the latter). Each time you jump to the next block, you need to make sure there's really a full block's worth of data to read. For example, if the block size is 64 rows, you need to check that at least 64 rows are left that you haven't read yet. If there aren't, then you can only read in as many as are left, and you'll get an error if you try to access more. Once you've worked your way to the end, you move to the next block of columns and start over, working through the rows. Again, you always need to make sure that you don't try to read more columns than exist in the raster.

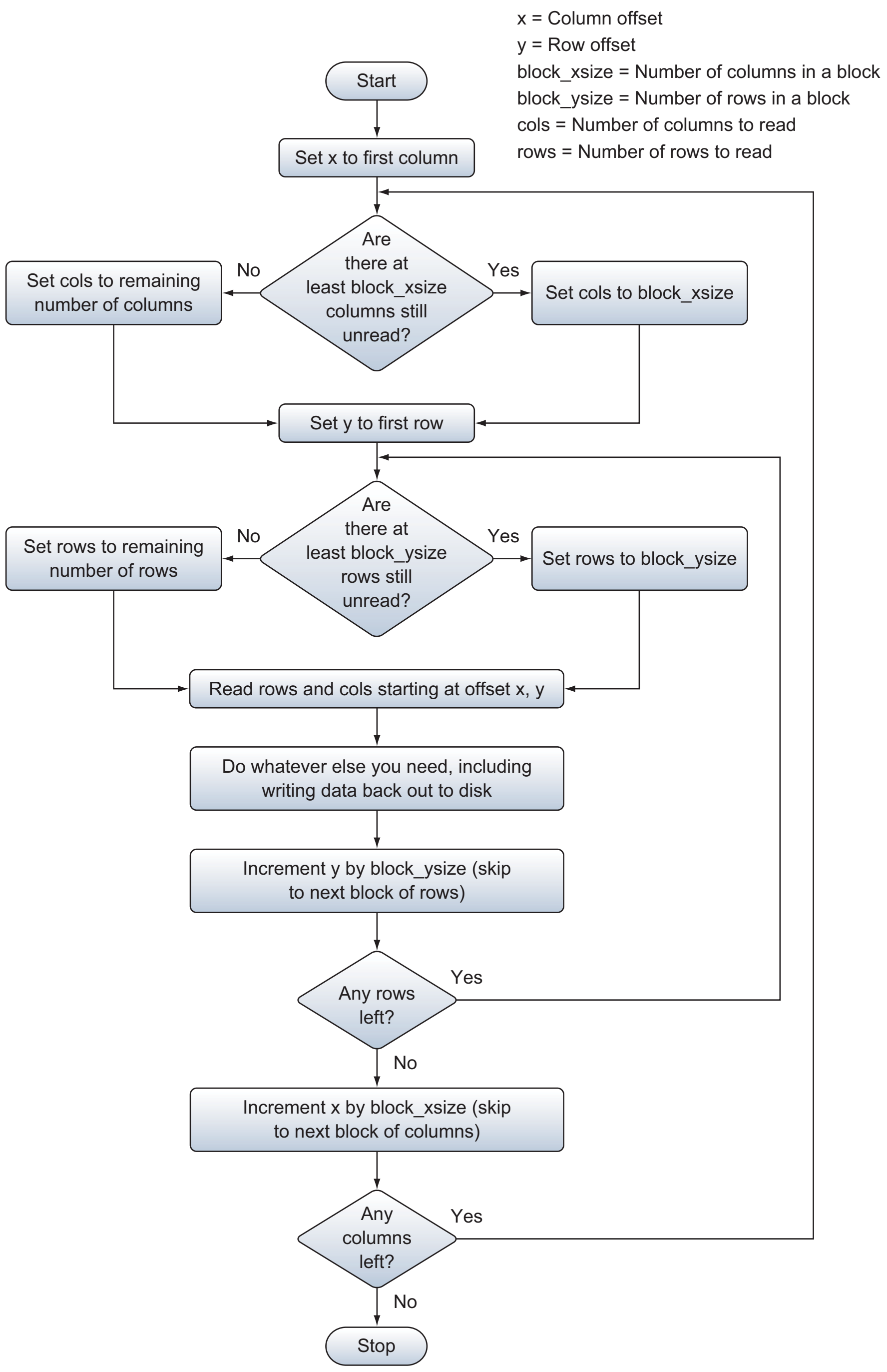
x = Column offset
y = Row offset
block_xsize = Number of columns in a block
block_ysize = Number of rows in a block
cols = Number of columns to read
rows = Number of rows to read
Start
Set x to first column
Are there at least block_xsize columns still unread?
No
Yes
Set cols to remaining number of columns
Set cols to block_xsize
Set y to first row
Are there at least block_ysize rows still unread?
No
Yes
Set rows to remaining number of rows
Set rows to block_ysize
Read rows and cols starting at offset x, y
Do whatever else you need, including writing data back out to disk
Increment y by block_ysize (skip to next block of rows)
Any rows left?
Yes
No
Increment x by block_xsize (skip to next block of columns)
Any columns left?
Yes
No
Stop

Figure 9.9 The process for reading and writing a raster block by block

Listing 9.2 shows how you might convert a digital elevation model from meters to feet, one block at a time. This is a small dataset, so you probably wouldn't need to break it up like this in the real world, but you would process a large dataset in the same way. This also shows you an example of dealing with `NoData` values in your raster, which are pixels that are considered to have a null value. Pixels must have a value, but a specific value can be specified as `NoData`, and therefore ignored.

Listing 9.2 Processing a raster by block

```
import os
import numpy as np
from osgeo import gdal

os.chdir(r'D:\osgeopy-data\Washington\dem')

in_ds = gdal.Open('gt30w140n90.tif')
in_band = in_ds.GetRasterBand(1)
xsize = in_band.XSize
ysize = in_band.YSize
block_xsize, block_ysize = in_band.GetBlockSize()      Get block size and
nodata = in_band.GetNoDataValue()                      NoData value

out_ds = in_ds.GetDriver().Create(
    'dem_feet.tif', xsize, ysize, 1, in_band.DataType)
out_ds.SetProjection(in_ds.GetProjection())
out_ds.SetGeoTransform(in_ds.GetGeoTransform())
out_band = out_ds.GetRasterBand(1)

for x in range(0, xsize, block_xsize):
    if x + block_xsize < xsize:                  Get number of
        cols = block_xsize                       columns to read
    else:
        cols = xsize - x
    for y in range(0, ysize, block_ysize):
        if y + block_ysize < ysize:                  Get number of
            rows = block_ysize                       rows to read
        else:
            rows = ysize - y
        data = in_band.ReadAsArray(x, y, cols, rows)          Read and write one
        data = np.where(data == nodata, nodata, data * 3.28084)   block's worth of data
        out_band.WriteArray(data, x, y)

out_band.FlushCache()
out_band.SetNoDataValue(nodata)                   Compute statistics at end,
out_band.ComputeStatistics(False)                 after setting NoData
out_ds.BuildOverviews('average', [2, 4, 8, 16, 32])
del out_ds
```

You can probably figure out what's happening at the beginning of this example. You open the dataset and get information about the band, including the size of its blocks and its `NoData` value. After creating the output dataset, you start looping through the blocks in the horizontal (x) direction. You start at column 0 and go up to the last column, represented with index xsize. The twist is that each time through the loop, you increment x by the number of columns in a block (the third argument to range is the amount to increment by), so you skip from the beginning of one block to the beginning of the next. Then you store the number of columns to read in a variable called cols. If there's a full block's worth of columns left to read, this variable is set to the number of columns in a block. But if there aren't enough columns, as would be the case when x is equal to 10 in figure 9.10, the number of remaining columns (three in the figure) is used instead. You need to do this because you'll get an error if you try to read more rows or columns than exist.

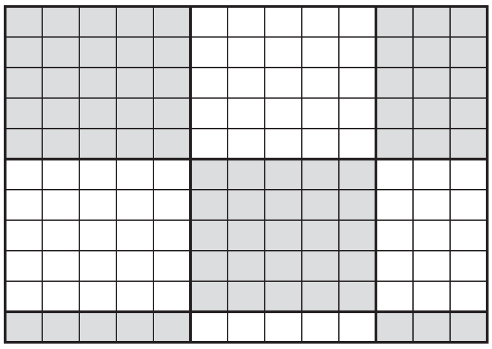
Figure 9.10 A small example image with a block size of five rows and five columns. Alternating blocks are shaded to make it easy to see. The upper-left pixel has offset 0,0.

After computing the number of columns to read, you repeat the process for the number of rows. As shown in figure 9.10, the first two times through that second loop you read five rows, but the third time there's only one row left to read. After processing the first five columns of all rows, you go to the next iteration of the outer loop and process the next five columns, and then the last three columns.

Once you figure out how many rows and columns to read, you pass those numbers, along with the current row and column offsets, to ReadAsArray to get a block's worth of data back:

```
data = in_band.ReadAsArray(x, y, cols, rows)
```

The next step is to convert the values, which come in as meters, to feet. You use the NumPy where function to help with this. This function is like an if-else statement. The first parameter is the condition to check, in this case whether or not the pixel value is equal to the NoData value. The second parameter is the output value if the condition is true. If the incoming pixel is NoData, you output NoData as well. The third parameter is the value to output if the condition is false, so this is where you convert the values to feet by multiplying them by 3.28084:

```
data = np.where(data == nodata, nodata, data * 3.28084)
```

After converting valid pixels to feet, you pass the data to WriteArray using the current row and column offsets before continuing on to the next block:

```
out_band.WriteArray(data, x, y)
```

After processing all of the blocks, you calculate statistics and build the overviews. To exclude the NoData pixels from the statistics calculation, you have to tell the band which value represents NoData before calling ComputeStatistics. You might be tempted to calculate statistics inside your loop, but you want the statistics to be based on all of the pixels in the band, so you need to wait until all of the pixel values have been calculated.

Obviously this method of looping through blocks is more complicated than reading and writing an entire band at once, but it's invaluable if you're low on RAM.

9.3.1 Using real-world coordinates

Until now, we've only considered pixel offsets when deciding where to start reading or writing data, but most of the time you'll have real-world coordinates instead. Fortunately, converting between the two is easy, as long as your coordinates use the same SRS as the raster. You saw earlier how to calculate coordinates of individual pixels, and now you need to reverse that process. All of the data required, including the origin coordinates, pixel sizes, and rotation values, are stored in the geotransform you've been copying between datasets. The geotransform is a tuple containing the six values shown in table 9.2. The rotation values are usually 0; in fact, I can't recall ever using an image that wasn't north up, but they're certainly out there.

Table 9.2 `GeoTransform` items

Index	Description
0	Origin x coordinate
1	Pixel width
2	x pixel rotation (0° if image is north up)
3	Origin y coordinate
4	y pixel rotation (0° if image is north up)
5	Pixel height (negative)

You could use this information to apply the affine transformation yourself, but GDAL provides a function that does it for you, called ApplyGeoTransform, that takes a geotransform, an x value, and a y value. When used with a dataset's geotransform, this function converts image coordinates (offsets) to real-world coordinates. But right now you're interested in going the other direction, so you need to get the inverse of the dataset's geotransform. Fortunately, a function exists for that, but you use it differently depending on the version of GDAL that you're using. If you're using GDAL 1.x, the InvGeoTransform function returns a success flag and a new geotransform that can be used to go the other direction:

```
gt = ds.GetGeoTransform()
success, inv_gt = gdal.InvGeoTransform(gt)
```

If all went well, then the success flag will be 1, but if the affine transformation couldn't be inverted, it returns 0 instead. Because of this, you should check the value of the success flag before continuing if you're not certain that the geotransform can be inverted.

If you're using GDAL 2.x, then the InvGeoTransform function only returns one item: a geotransform if one could be calculated, or None if not. In this case, you need to make sure that the returned value isn't equal to None:

```
inv_gt = gdal.InvGeoTransform(gt)
```

Now that you have an inverse geotransform, you can use it to convert real-world coordinates to image coordinates. For example, say you need the pixel value at coordinates 465200, 5296000. The following code would get it, assuming that the raster covers that location:

```
offsets = gdal.ApplyGeoTransform(inv_gt, 465200, 5296000)
xoff, yoff = map(int, offsets)
value = band.ReadAsArray(xoff, yoff, 1, 1)[0,0]
```

The ApplyGeoTransform function returns an x and a y value as floating-point numbers, but you need integer offsets to pass to ReadAsArray. If you forget to convert the offsets to integers, you'll get an error. After getting the integers, you read in one row and one column starting at those offsets. You might think that this would return a number, but not quite. Remember that ReadAsArray returns a two-dimensional array, and it does this even for only one row and/or one column. To get the actual pixel value, you still have to get the value in the first row and first column (position [0,0]) in the array.

This method is extremely inefficient if you need to sample pixel values at many different locations, however. In that case, you're better off reading in the entire band and then pulling the appropriate values from that array. This is because read and write operations are expensive, so doing a new read operation for each point is much slower than doing one large read operation for the whole band. The code to get the same pixel value using this method might look like this:

```
data = band.ReadAsArray()
x, y = map(int, gdal.ApplyGeoTransform(inv_gt, 465200, 5296000))
value = data[yoff, xoff]
```

Obviously, you wouldn't read the whole band in for each point; you'd do that once but then repeat the last two lines for each point. Notice that the row and column offsets are reversed when pulling the pixel value from the NumPy array, because NumPy wants offsets as [row, column], not [x, y] (which is the same as [column, row]).

TIP Use [row, column] offsets for NumPy arrays. This is the reverse of what you're used to using with GDAL.

The ability to convert between real-world coordinates and offsets is also important if you want to extract a spatial subset and save it to a new image, because you need to change the origin coordinates in the geotransform. Say you wanted to extract Vashon Island (figure 9.11) out of the natural color Landsat image you created earlier, and

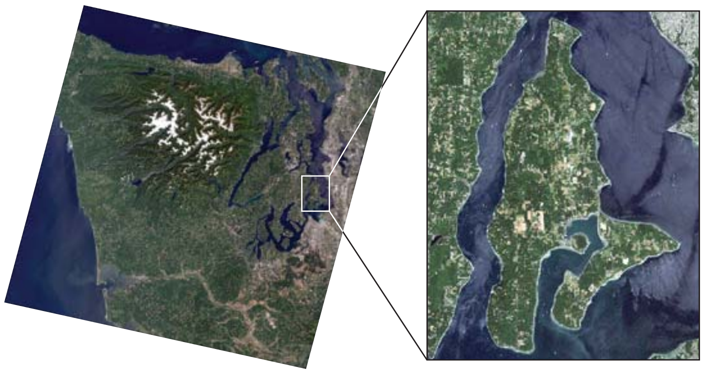
Figure 9.11 The goal of listing 9.3 is to extract Vashon Island out of the natural color Landsat image created earlier.

you're given the upper-left and lower-right coordinates of the area of interest. You need to convert these into pixel offsets so you know what data to read, but it's unlikely that these bounding coordinates correspond exactly to pixel boundaries, so you also need to find the true upper-left coordinates for the subset you extracted. The following listing shows an example of this.

Listing 9.3 Extracting and saving a subset of an image

```
import os
from osgeo import gdal

vashon_ulx, vashon_uly = 532000, 5262600
vashon_lrx, vashon_lry = 548500, 5241500

os.chdir(r'D:\osgeopy-data\Landsat\Washington')
in_ds = gdal.Open('nat_color.tif')
in_gt = in_ds.GetGeoTransform()

inv_gt = gdal.InvGeoTransform(in_gt)
if gdal.VersionInfo()[0] == '1':
    if inv_gt[0] == 1:
        inv_gt = inv_gt[1]
    else:
        raise RuntimeError('Inverse geotransform failed')
elif inv_gt is None:
    raise RuntimeError('Inverse geotransform failed')

offsets_ul = gdal.ApplyGeoTransform(
    inv_gt, vashon_ulx, vashon_uly)
offsets_lr = gdal.ApplyGeoTransform(
    inv_gt, vashon_lrx, vashon_lry)
off_ulx, off_uly = map(int, offsets_ul)
off_lrx, off_lry = map(int, offsets_lr)

rows = off_lry - off_uly
columns = off_lrx - off_ulx
```

Compute upper-left and lower-right offsets

Compute number of rows and columns to extract

```
gtiff_driver = gdal.GetDriverByName('GTiff')
out_ds = gtiff_driver.Create('vashon2.tif', columns, rows, 3)
out_ds.SetProjection(in_ds.GetProjection())
subset_ulx, subset_uly = gdal.ApplyGeoTransform(
    in_gt, off_ulx, off_uly)
out_gt = list(in_gt)
out_gt[0] = subset_ulx
out_gt[3] = subset_uly
out_ds.SetGeoTransform(out_gt)

for i in range(1, 4):
    in_band = in_ds.GetRasterBand(i)
    out_band = out_ds.GetRasterBand(i)
    data = in_band.ReadAsArray(
        off_ulx, off_uly, columns, rows)
    out_band.WriteArray(data)

del out_ds
```

Put new origin coordinates in geotransform

Read in data using computed values

Write out data starting at the origin

You've seen everything in this example before, but you haven't seen it put together quite like this. The important parts are where you compute the offsets for the upper-left and lower-right corners of Vashon Island, based on the coordinates at the top of the script (in real life you probably wouldn't want the coordinates hardcoded in, but it works for the example). Then you subtract the upper-left offsets from the lower-right offsets to get the total numbers of rows and columns to extract.

Once you have that basic information, you create an output image with these new dimensions, rather than the dimensions of the original image. The projection information is copied over unchanged, but you have to alter the geotransform to reflect the upper-left coordinates of the subset. You can't use the upper-left coordinates that you calculated because those probably fall in the middle of a pixel somewhere, but you need the coordinates of the pixel corner. Notice that you use the original geotransform for this, not the inverted one, because you're converting offsets to real-world coordinates. Then, because the geotransform is returned as a tuple, you have to convert it to a list before you can insert the new upper-left coordinates.

After all that housekeeping, you copy the data from the original to the new image. You start reading at the upper-left offsets, and grab the numbers of columns and rows that you computed earlier. There's no reason to provide offsets when writing the data out because the new image will only contain the subset, so you want to start writing at the origin.

You'd probably also compute statistics and build overviews, but those steps aren't absolutely necessary, so I left them out in the interest of space.

9.3.2 Resampling data

A nice feature of the `ReadAsArray` function is that you can use it to resample data as it's read in, either by specifying the output buffer size or passing an existing buffer array. As a reminder, here's what the function signature looks like:

```
band.ReadAsArray([xoff], [yoff], [win_xsize], [win_ysize], [buf_xsize],
                 [buf_ysize], [buf_obj])
```

The win parameters specify the number of rows and columns to read from the band, and the buf parameters specify the size of the array to put those pixel values into. An array with larger dimensions than the original will resample to smaller pixels, while one with smaller dimensions will resample to larger pixels using nearest-neighbor interpolation.

RESAMPLING TO SMALLER PIXELS

To resample data to a finer resolution, provide an array that's larger than the data being read in so that the pixel values need to be repeated to fill the target array. For example, this will create four pixels for every one pixel, essentially cutting the pixel size in half, as shown in figure 9.12:

```
band.ReadAsArray(1400, 6000, 3, 2, 6, 4)
```

This works because you're reading three columns and two rows from the band, but putting that data into an array with six columns and four rows, so each row and column is duplicated to fill the output array.

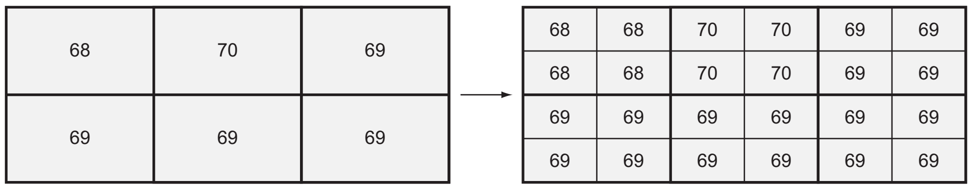

Figure 9.12 Pixel values are repeated four times each when the numbers of rows and columns are doubled.

This is all well and good, but how do you deal with the new cell size if you need to write the data out to a new image? It's easy, because all you have to do is alter the geotransform so that it specifies a smaller pixel size. Take a look at the following listing to see how you might resample an entire image to a smaller pixel size.

Listing 9.4 Resample an image to a smaller pixel size

```
import os
from osgeo import gdal

os.chdir(r'D:\osgeopy-data\Landsat\Washington')

in_ds = gdal.Open('p047r027_7t20000730_z10_nn10.tif')
in_band = in_ds.GetRasterBand(1)
out_rows = in_band.YSize * 2                        Get number of output
out_columns = in_band.XSize * 2                     rows and columns

gtiff_driver = gdal.GetDriverByName('GTiff')
out_ds = gtiff_driver.Create('band1_resampled.tif',      Create output dataset
    out_columns, out_rows)
```

```
out_ds.SetProjection(in_ds.GetProjection())
geotransform = list(in_ds.GetGeoTransform())
geotransform [1] /= 2
geotransform [5] /= 2
out_ds.SetGeoTransform(geotransform)

data = in_band.ReadAsArray(
    buf_xsize=out_columns, buf_ysize=out_rows)
out_band = out_ds.GetRasterBand(1)
out_band.WriteArray(data)

out_band.FlushCache()
out_band.ComputeStatistics(False)
out_ds.BuildOverviews('average', [2, 4, 8, 16, 32, 64])
del out_ds
```

Edit the geotransform so pixels are one-quarter previous size

Specify a larger buffer size when reading data

Build appropriate number of overviews for larger image

This example has a few important things to note. First, you double the number of rows and columns when creating the new dataset, and you pass these same numbers as parameters to `ReadAsArray`. This ensures that your input data dimensions match your output data dimensions, and also causes the data to be resampled to those larger dimensions. Instead of using the `buf_xsize` and `buf_ysize` parameters, you could have used an existing array for the `buf_obj` parameter and gotten the same results. You could also have provided the `win_xsize` and `win_ysize` parameters, but they default to the original numbers of rows and columns, which is what you want.

You also edit the geotransform to reflect the smaller pixel size. The second item in the geotransform is the pixel width, and the sixth is the pixel height, so you divide each of those values by two and overwrite the original values. Because this image still covers the same spatial extent as the original, you don't need to change any of the other values. Once you finish editing, you set the geotransform onto the new dataset. Fortunately, editing the geotransform doesn't alter the geotransform for the original image because the tuple isn't linked to the dataset, so you're not introducing any complications there.

If you hadn't changed the pixel size and instead copied the original geotransform to the new dataset, your output would have looked like the larger image shown in figure 9.13. As you may recall, the spatial extent of a raster is determined from the origin coordinates and the pixel size. The upper-left corner coordinates would be the same, but the incorrect pixel size would cause the image to cover twice the distance in each direction. In this case, a satellite image of northwestern Washington State would appear to extend into eastern Washington and south into Oregon, which is obviously incorrect.

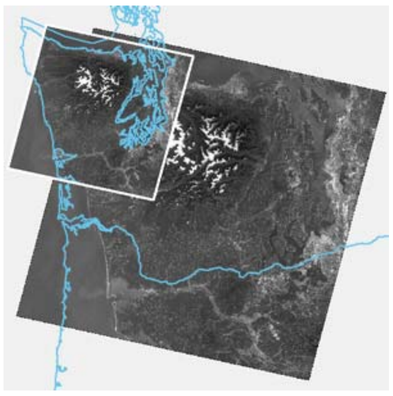
Figure 9.13 This illustrates the result of resampling to a smaller pixel size without changing the size in the geotransform. The smaller image in the upper left is correct. The larger one was created by using the unedited geotransform from the input image.

It should be clear by now how important an accurate geotransform is. If a raster appears to be in the wrong location or the wrong size when opened in a GIS, then an incorrect geotransform is a likely culprit, as is an erroneous spatial reference system.

RESAMPLING TO LARGER PIXELS

You can also resample to a coarser resolution by providing a smaller buffer array when reading the data. In this case, one pixel takes the place of several cells, and nearest-neighbor interpolation is used to determine which value is used (figure 9.14). The following example replaces four pixels with one:

```
data = np.empty((2, 3), np.int)
band.ReadAsArray(1400, 6000, 6, 4, buf_obj=data)
```

Here, an empty integer NumPy array with three columns and two rows is created beforehand and then passed as an argument to `ReadAsArray`. The six columns and four rows requested from the image are resampled to fit into this smaller array. By the way, you don't need to catch the return value from `ReadAsArray` in this case, because you already have the `data` variable. But not only is the `data` variable filled automatically, it's also returned from the function, so you can grab it that way if you'd like, but it's not necessary.

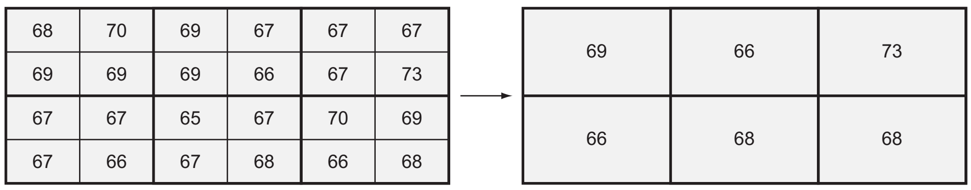

Figure 9.14 Nearest-neighbor interpolation is used to select a pixel value when resampling to smaller dimensions. In this case, the lower-right pixel value for each block of four pixels is used in the output.

Although this technique usually uses nearest-neighbor interpolation to resample, if you have an overview layer of the requested resolution, then that will be used instead. If the appropriate overview was built with average interpolation, then that's what you'd get when using `ReadAsArray,` rather than nearest-neighbor.

As with resampling to smaller pixels, you need to change the pixel size in the geotransform when writing the data back out to another dataset. The only difference is that in this case, you want to decrease the number of rows and columns and increase the pixel size. You could alter listing 9.4 to resample to coarser pixels by dividing the rows and columns by 2 instead of multiplying, and multiplying instead of dividing the pixel size. You could also get away with building fewer overview layers. Other than that, the technique is the same. If you forget to change the pixel size, then you end up with an image that's compressed into too small of an area instead of the too large area shown back in figure 9.13.

9.4 Byte sequences

If you've looked through appendix E, you've probably noticed that ReadAsArray and WriteArray aren't the only ways to read and write data with GDAL. (Appendixes C through E are available online on the Manning Publications website at https://www.manning.com/books/geoprocessing-with-python.) You can also read data into a sequence of Python bytes, which is much like a string made up of the ASCII codes corresponding to the numeric pixel values. Unlike strings, byte sequences can't be modified, although you'll learn how to get around that in this section. This is a bit faster than converting to a NumPy array, but I prefer to get an array so that I can manipulate it mathematically. But if you'd like to use bytes instead, or don't need to manipulate the data, the parameters for ReadRaster are similar to those for ReadAsArray. Here's the signature for the dataset version of ReadRaster:

```
ReadRaster([xoff], [yoff], [xsize], [ysize], [buf_xsize], [buf_ysize],
           [buf_type], [band_list], [buf_pixel_space], [buf_line_space],
           [buf_band_space])
```

- xoff is the column to start reading at. The default value is 0.
- yoff is the row to start reading at. The default value is 0.
- xsize is the number of columns to read. The default is to read them all.
- ysize is the number of rows to read. The default is to read them all.
- buf_xsize is the number of columns in the returned sequence. The default is to use the xsize value. Data will be resampled if this value is different than xsize.
- buf_ysize is the number of rows in the returned sequence. The default is to use the ysize value. Data will be resampled if this value is different than ysize.
- buf_type is the target GDAL data type for the returned sequence. The default is the same as the original data.
- band_list is a list of band indices to read. The default is to read all bands.
- buf_pixel_space is the byte offset between pixels in the sequence. The default is the size of buf_type.
- buf_line_space is the byte offset between lines in the sequence. The default is the size of buf_type multiplied by xsize.
- buf_band_space is the byte offset between bands in the sequence. The default is the size of buf_line_space multiplied by ysize.

The first six parameters are the same as for ReadAsArray. The buf_type parameter is a GDAL data type constant from table 9.1 and is used to specify the data type used for the byte sequence. This can be used to change the data type as it's read in. For example, if the raster is of type byte, but you provide GDT_float32 for this parameter, then the resulting byte string will represent the pixel values as floating-point instead of byte. You can also provide a list of bands to read, and they'll be returned in the order you specify. You can even include a band more than once, although I'm not sure why you'd want to. The last three parameters change the spacing of data in the returned byte string and can be used to work with unusually interleaved datasets, but chances are that you'll never need these. The parameters for the band version of `ReadRaster` are the same, except that `band_list` and `buf_band_space` are missing.

Anyway, if you were to print out the results of a call to `ReadRaster`, the result would be something like `b'\x1c\x1d\x1c\x1e'`, which doesn't mean a whole lot to me. You can access elements by index, however, and those will look more familiar. Byte strings are immutable, which means they can't change, but you can convert them to byte arrays if you need to edit values. The following interactive session shows you an example of this:

```
>>> data = ds.ReadRaster(1400, 6000, 2, 2, band_list=[1])
>>> data
b'\x1c\x1d\x1c\x1e'
>>> data[0]
28
>>> bytearray_data = bytearray(data)
>>> bytearray_data[0] = 50
>>> bytearray_data[0]
50
```

You can also convert a byte string to a tuple using the built-in struct module. Here you need to provide a format string that specifies what type, and how many, elements are contained in the string. In this example, you're using a format string such as "BBBB" to specify four bytes. See the Python struct documentation for other formats.

```
tuple_data = struct.unpack('B' * 4, data)
```

If you want to turn the byte string into a NumPy array, you can do that using the tuple from `unpack`, or by using the NumPy `fromstring` function to convert the byte string directly (although if you want a NumPy array, maybe you should use `ReadAsArray`). As with using `unpack`, you have to provide the data type that the sequence uses when converting it to a NumPy array. Both of these methods return a one-dimensional array, so you'll have to reshape it to multidimensional if needed. Examples of these operations are shown here:

```
numpy_data1 = np.array(tuple_data)
numpy_data2 = np.fromstring(data, np.int8)
reshaped_data = np.reshape(numpy_data2, (2,2))
```

The parameters for writing data from byte strings are similar to those for reading, although the first five arguments are required instead of optional:

```
def WriteRaster(xoff, yoff, xsize, ysize, buf_string, [buf_xsize],
                [buf_ysize], [buf_type], [band_list], [buf_pixel_space],
                [buf_line_space], [buf_band_space])
```

- `xoff` is the column to start writing at.
- `yoff` is the row to start writing at.
- `xsize` is the number of columns to write.
- `ysize` is the number of rows to write.

- `buf_string` is the byte sequence to write.
- `buf_xsize` is the number of columns in the byte sequence. The default is to use the `xsize` value. Data will be resampled if this value is different than `xsize`.
- `buf_ysize` is the number of rows in the byte sequence. The default is to use the `ysize` value. Data will be resampled if this value is different than `ysize`.
- `buf_type` is the GDAL data type of the byte sequence. The default is the same as the dataset being written to.
- `band_list` is a list of band indices to write. The default is to write all bands.
- `buf_pixel_space` is the byte offset between pixels in the byte sequence. The default is the size of `buf_type`.
- `buf_line_space` is the byte offset between lines in the byte sequence. The default is the size of `buf_type` multiplied by `xsize`.
- `buf_band_space` is the byte offset between bands in the byte sequence. The default is the size of `buf_line_space` multiplied by `ysize`.

Once again, the band version is the same, except that the `band_list` and `buf_band_space` parameters don't exist.

You could write a byte sequence, called `data`, that contains six columns and four rows out to a dataset like this:

```
ds.WriteRaster(1400, 6000, 6, 4, data, band_list=[1])
```

Let's try resampling an image to a larger pixel size using bytes instead of NumPy arrays.

Listing 9.5 Resample an image to a larger pixel size using byte sequences

```
import os
import numpy as np
from osgeo import gdal

os.chdir(r'D:\osgeopy-data\Landsat\Washington')

in_ds = gdal.Open('nat_color.tif')
out_rows = int(in_ds.RasterYSize / 2)                    Get number of output
out_columns = int(in_ds.RasterXSize / 2)                 rows and columns
num_bands = in_ds.RasterCount

gtiff_driver = gdal.GetDriverByName('GTiff')
out_ds = gtiff_driver.Create('nat_color_resampled.tif',
    out_columns, out_rows, num_bands)                       Create output dataset

out_ds.SetProjection(in_ds.GetProjection())
geotransform = list(in_ds.GetGeoTransform())
geotransform[1] *= 2                                   Edit the geotransform
geotransform[5] *= 2                                   so pixel sizes are larger
out_ds.SetGeoTransform(geotransform)

data = in_ds.ReadRaster(                                 Use a smaller buffer
    buf_xsize=out_columns, buf_ysize=out_rows)           to read and write data
```

```
out_ds.WriteRaster(0, 0, out_columns, out_rows, data)
out_ds.FlushCache()
for i in range(num_bands):
    out_ds.GetRasterBand(i + 1).ComputeStatistics(False)

out_ds.BuildOverviews('average', [2, 4, 8, 16])
del out_ds
```

Build appropriate number of overviews for smaller image

In many ways this example is similar to listing 9.4, except that the numbers of output rows and columns are halved instead of doubled, and the pixel size is doubled instead of halved. Notice that in this case you need to ensure that the numbers of rows and columns are integers, because the result of the division might be floating-point, and the dataset creation function doesn't like that.

The interesting part is where you read and write the data. Because all rows, columns, and bands are read by default, you didn't have to do anything about those. But because you want the data resampled into half as many rows and columns, you pass these smaller numbers in using the `buf_xsize` and `buf_ysize` parameters. This causes the data to be resampled as GDAL reads it into the byte sequence. Then you write the data out to the new dataset starting at the first row and column. You also tell `WriteRaster` how many rows and columns are contained in the byte sequence, because unlike a NumPy array, this isn't obvious. A byte sequence that is 32 bytes long might contain one 32-bit integer, or it might contain four 8-bit integers. Although `WriteRaster` can figure out how many bytes are in the sequence, it doesn't know how to convert those to pixel values until you tell it how many values there are supposed to be.

9.5 Subdatasets

Several types of datasets can contain other datasets, which each in turn contain bands (figure 9.15). One example of this is the MODIS imagery distributed by the United States Geological Service, which comes as a hierarchical data format (HDF) file. If your dataset contains subdatasets, you can get a list of them with the `GetSubDatasets` function and then use that information to open the one you want.

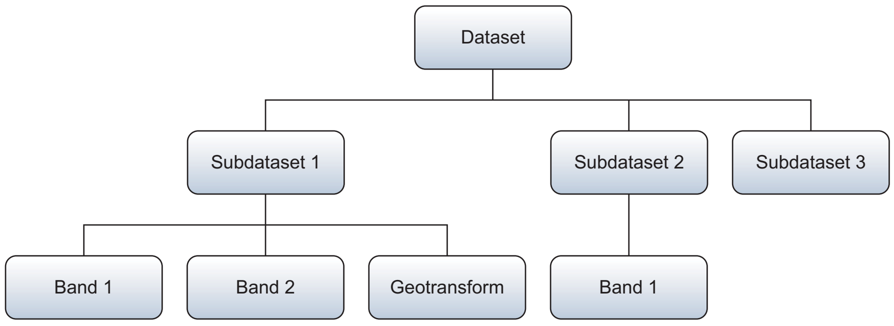

Figure 9.15 Several types of datasets include subdatasets. Each of these subdatasets is structured like a normal dataset and contains its own bands and georeferencing information.

As an example, let's open a subdataset contained in a MODIS file. Note that the HDF driver isn't included in GDAL by default, so this example won't work for you if your GDAL version doesn't include HDF support. Assuming you can work with HDF files, the first step is to open the HDF file as a dataset:

```
ds = gdal.Open('MYD13Q1.A2014313.h20v11.005.2014330092746.hdf')
```

Now you can get the list of subdatasets contained in this open dataset. The `GetSubDatasets` method returns a list of tuples, with one tuple per subdataset. Each tuple contains the name and the description of the subdataset, in that order. The following snippet gets this list and then prints out the name and description for each of the subdatasets:

```
subdatasets = ds.GetSubDatasets()
print('Number of subdatasets: {}'.format(len(subdatasets)))
for sd in subdatasets:
    print('Name: {0}\nDescription:{1}\n'.format(*sd))
```

The first few lines of output look like this, and show the information for the first subdataset, which is the NDVI (normalized difference vegetation index), but there are 11 more not shown:

```
Number of subdatasets: 12
Name: HDF4_EOS:EOS_GRID:"MYD13Q1.A2014313.h20v11.005.2014330092746.hdf":
➥ MODIS_Grid_16DAY_250m_500m_VI:250m 16 days NDVI
Description:[4800x4800] 250m 16 days NDVI MODIS_Grid_16DAY_250m_500m_VI
➥ (16-bit integer)
```

To open a subdataset, pass its name to `gdalOpen`. For example, this gets the tuple corresponding to the first subdataset, gets the first item (the name) from the tuple, and then uses that to open the subdataset:

```
ndvi_ds = gdal.Open(subdatasets[0][0])
```

Similarly, you would use `subdatasets[4][0]` to open the fifth subdataset. Once you've opened a subdataset like this, it can be treated like any other dataset. For example, you could get the first band in the NDVI subdataset using `ndvi_ds.GetRasterBand(1)`.

9.6 Web map services

Let's take a quick look at web map services, which are used to serve images across the web for things like basemaps. We'll try out an OGC web map service that creates an image based on your request, but you have other methods of accessing basemaps. For example, both OpenStreetMap and Google use pre-rendered tiled images. To use those, you need to know the tile that you want, and nothing is rendered on the fly (well, it could be, depending on how the images are cached on the server, but the idea is that the tiles already exist so they provide fast access).

GDAL allows you to use XML files to specify the parameters for a map service, and all of the possibilities are documented at http://www.gdal.org/frmt_wms.html.

The following listing shows the XML describing an imagery basemap from the US National Map.

Listing 9.6 XML describing a web map service

```
<GDAL_WMS>
    <Service name="WMS">
        <Version>1.3.0</Version>
        <ServerURL>http://raster.nationalmap.gov/arcgis/services/
        ➥ Orthoimagery/USGS_EROS_Ortho_1Foot/ImageServer/WMSServer?
        ➥ </ServerURL>
        <CRS>CRS:84</CRS>
        <ImageFormat>image/png</ImageFormat>
        <Layers>0</Layers>
    </Service>
    <DataWindow>
        <UpperLeftX>-74.054444</UpperLeftX>
        <UpperLeftY>40.699167</UpperLeftY>
        <LowerRightX>-74.034444</LowerRightX>
        <LowerRightY>40.679167</LowerRightY>
        <SizeX>300</SizeX>
        <SizeY>300</SizeY>
    </DataWindow>
    <BandsCount>4</BandsCount>
</GDAL_WMS>
```

You need to know certain information about the service to create an XML specification, however. OGC web map services allow you to request information about them using a GetCapabilities request. If you don't know the base URL for the service, you're out of luck, but assuming you do know it, tack "?request=GetCapabilities&service=WMS" onto the end and view the result in a browser. For example, the URL for the service defined in listing 9.6 is http://raster.nationalmap.gov/arcgis/services/Orthoimagery/USGS_EROS_Ortho_1Foot/ImageServer/WMSServer?request=GetCapabilities&service=WMS.

This is a lot of information, but we'll focus on a few parts that are important for the Service section of the XML. Look at the first line of output:

```
<WMS_Capabilities xmlns=http://www.opengis.net/wms
➥ xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="1.3.0"
➥ xsi:schemaLocation="http://www.opengis.net/wms
➥ http://schemas.opengis.net/wms/1.3.0/capabilities_1_3_0.xsd">
```

Part of that line specifies the WMS version as 1.3.0. Add that information to the Version section of your XML. Now look through the GetCapabilities results until you find the GetMap section. The first part of it looks like this:

```
<GetMap>
    <Format>image/tiff</Format>
    <Format>image/png</Format>
    <Format>image/png24</Format>
    <Format>image/png32</Format>
    <Format>image/bmp</Format>
    <Format>image/jpeg</Format>
    <Format>image/svg</Format>
    <Format>image/bil</Format>
```

These are the formats that the service can provide, and you should include one of them in the ImageFormat section of your XML. Now look for the Layer section in the GetCapabilities output. Here are the first few lines of that section:

```
<Layer>
    <Name>0</Name>
    <Title>USGS_EROS_Ortho_1Foot</Title>
    <Abstract>
        The USGS_EROS_Ortho_1Foot service from The National Map contains 1
  foot orthoimagery, and is viewable at all scales.
    </Abstract>
```

We want to use the layer called USGS_EROS_Ortho_1Foot, but the Name value is the important one. In this case, the name is "0," which isn't too descriptive, but it's what you need to add to the Layer section of the XML. If you keep looking at the Layer section of the capabilities, you'll see a lengthy list of CRS values, which are the coordinate systems supported by the service. Here are the first few:

```
    <CRS>CRS:84</CRS>
    <CRS>EPSG:4326</CRS>
    <CRS>EPSG:3857</CRS>
```

You guessed it. Select one of these for your output and add it to the CRS section of your XML.

Now that the service is defined in your XML, you need to specify the geographic extent that you want to retrieve. You do this with the DataWindow section. The UpperLeftX, UpperLeftY, LowerRightX, and LowerRightY are the minimum x, maximum y, maximum x, and minimum y values, respectively. The SizeX and SizeY parameters specify the number of columns and rows for the output image.

Once you have your XML saved, pass the filename to the GDAL `Open` function, and if everything is configured correctly, it will be opened as a dataset. At this point you could get the bands and read the data into an array, or you could save the image to a local file using `CreateCopy`. For example, this snippet uses the XML from listing 9.6 to save a local image of Liberty Island in New York Harbor (figure 9.16):

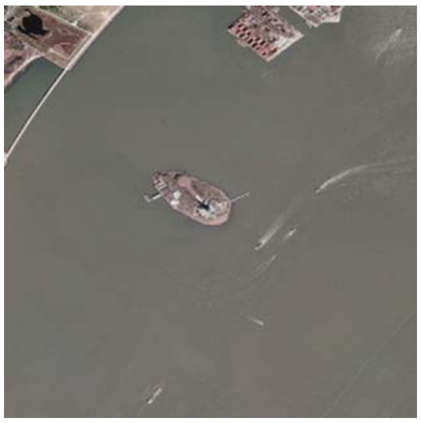
Figure 9.16 An image of Liberty Island in New York Harbor obtained using an OGC web map service

```
ds = gdal.Open('listing9_6.xml')
gdal.GetDriverByName('PNG').CreateCopy(r'D:\Temp\liberty.png', ds)
```

If you need to request images with different spatial extents or other parameters that regularly change, it would make sense to create an XML template and format it with the desired values when required.

9.7 Summary

- Raster datasets are ideal for continuous data without sharp boundaries, such as elevation, precipitation, or satellite imagery.
- In the interest of disk space, don't use smaller pixel sizes or larger data types than necessary.
- If you need to use your data for analysis, be sure to use a lossless compression algorithm or no compression at all.
- Use overviews for rapid display of raster data.
- Always use nearest-neighbor resampling for non-continuous raster data because other methods will result in new pixel values that don't correspond to the originals.
- For best performance, make as few read/write calls as possible, but don't try to keep more data in memory than you have RAM.
- Don't forget to edit the geotransform if you change spatial extent or pixel size.
- Don't try to read or write past the edge of an image.
- Use the buffer parameters to resample data while reading or writing.
- Use `ReadAsArray` if you want to use NumPy to manipulate your data in memory, but `ReadRaster` is slightly faster if you only need to copy data.