11 Map algebra with NumPy and SciPy

This chapter covers

- Manipulating data with NumPy
- Using NumPy and SciPy for local, focal, and zonal map algebra calculations
- Using GDAL for global map algebra calculations
- Resampling data

You've seen how to read and write raster data, but you still don't know how to manipulate pixel values to do any analysis. Aerial photos make nice basemaps, but many types of raster datasets are used for scientific data analysis. For example, you'll see several examples of landcover classification in the next chapter. If you wanted to create your own landcover model, you might collect satellite imagery, elevation data, and climate data such as average precipitation or temperature, all of which are generally raster datasets. If you wanted to use vector data in the model, such as soil types, you'd convert it to raster first so that you could use it with your raster datasets. You could then use techniques from this chapter to derive slope and aspect from your elevation data and to combine all of your datasets to create a landcover model.

You'll learn several techniques for manipulating raster data in this chapter. For example, you'll learn how to apply calculations on a pixel-by-pixel basis to two or

more rasters. You'll also see how to use a small set of neighboring pixels to come up with new values. This is what happens when you smooth or sharpen a digital photo. Other calculations use all pixels in a raster, or divide them up based on some common value. Each of these has different uses, and you'll see examples of all of them.

If you plan on working with large raster datasets with Python, you need to be familiar with the SciPy project, which is a collection of Python modules designed for scientific computing. The NumPy, SciPy, and matplotlib modules are part of this. NumPy was designed to handle large arrays of data, which is perfect for raster data because a band is essentially a two-dimensional array of pixel values. SciPy contains routines for several kinds of scientific analysis, and it uses NumPy to hold the data. We'll look at both of these modules, along with a couple of others, in the next two chapters. Matplotlib is a plotting module that's also part of the SciPy project, and we'll look at it in the last chapter.

11.1 Introduction to NumPy

Entire books have been written about NumPy, but we'll take a brief look at how to create arrays and access specific values. When you use the GDAL `ReadAsArray` function, the data are put into a NumPy array for you. Once there, you can manipulate your data in many different ways. The bulk of this chapter discusses map algebra, which involves calculations on one or more arrays, and many of the examples won't make much sense if you don't understand the basics of working with NumPy arrays. For more in-depth information, please look at another book or refer to the excellent documentation online at http://www.numpy.org.

Accessing individual cell values in NumPy arrays is much the same as accessing values in a Python list, except that they have an index for each dimension of the array. For example, if you have a two-dimensional array, you need to provide both the row and the column offsets to specify a particular cell. Let's look at this and other basics from inside a Python interactive session.

> **TIP** By convention, the `numpy` module is renamed to `np` when importing it into a Python script. You don't have to follow suit, but you'll find that many examples, including those on the NumPy website, do this.

First create an example array using the `arange` function, which returns an array containing a sequence of numbers. Because this array only has one dimension, you can access elements with a single index. You can also get a slice, or section, of an array by providing starting and ending indices separated by a colon.

```
>>> import numpy as np
>>> a = np.arange(12)
>>> a
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
>>> a[1]
1
>>> a[1:5]
array([1, 2, 3, 4])
```

As long as the total number of elements in an array doesn't change, you can reshape it to different dimensions. For example, the array you created contains 12 elements, so it can be reshaped into a two-dimensional array with three rows and four columns because that also contains 12 elements. It couldn't be reshaped into an array with four rows and four columns, however, because that would require 16 elements. A two-dimensional array requires a row and a column index, in that order, to access its elements:

```
>>> a = np.reshape(a, (3,4))
>>> a
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> a[1,2]
6
```

TIP When specifying the shape of an array in a function, be sure to pass the dimensions in a tuple instead of as individual values.

If you only provide one index, `n`, it returns the entire n^{th} row. You can get an entire column by using a colon as the row index, which is the same as `0:n`, where `n` is the number of rows. You could retrieve the entire second row or third column like this:

```
>>> a[1]
array([4, 5, 6, 7])
>>> a[:,2]
array([ 2,  6, 10])
```

You can access a two-dimensional slice by providing starting and ending indices for both dimensions. Again, not providing a starting index on the left of the colon is the same as using 0, and if you don't provide an ending index, then you get the rest of the values in that dimension. You can also use negative numbers to leave rows or columns off the end.

```
>>> a[1:,1:3]
array([[ 5,  6],
       [ 9, 10]])
>>> a[2,:-1]
array([ 8,  9, 10])
```

Working with NumPy arrays is more than accessing cell values, though. You need to use multiple arrays together to implement many types of analyses. If two or more arrays have the same dimensions, you can perform mathematical and logical operations on them. These work on a cell-by-cell basis, so, for example, if you add two arrays, the [n, m] cell in the first array is added to the [n, m] cell in the second array. The same rule applies to operations such as multiplication; if you want mathematical matrix algebra behavior instead, use the `numpy.linalg` submodule.

```
>>> a = np.array([[1, 3, 4], [2, 7, 6]])
>>> b = np.array([[5, 2, 9], [3, 6, 4]])
>>> a
array([[1, 3, 4],
       [2, 7, 6]])
```

```
>>> b
array([[5, 2, 9],
       [3, 6, 4]])
>>> a + b
array([[ 6,  5, 13],
       [ 5, 13, 10]])
>>> a > b
array([[False,  True, False],
       [False,  True,  True]], dtype=bool)
```

Many different functions exist for working with arrays, including one that works much like an if-else statement. For example, you could create an array with certain values based on the comparison of two existing arrays like this:

```
>>> np.where(a > b, 10, 5)
array([[ 5, 10,  5],
       [ 5, 10, 10]])
```

The first parameter to the `where` function is the condition to check, the second parameter is the value to use if the condition is true, and the third is the value to use otherwise. These values can also be arrays, as long as they're the same size as the condition array. For example, you could get the larger of the two values at each location like this:

```
>>> np.where(a > b, a, b)
array([[5, 3, 9],
       [3, 7, 6]])
```

Now that you've seen these examples, let's look at another way to extract data from arrays. You aren't limited to one value or slices of contiguous data, because you can also use a list of indices. As an example, create an array of 12 random integers between 0 and 20 and then extract the ninth, first, and fourth values, in that order:

```
>>> a = np.random.randint(0, 20, 12)
>>> a
array([16, 16, 18,  1, 14,  2, 18, 19,  2, 16, 10,  8])
>>> a[[8, 0, 3]]
array([ 2, 16,  1])
```

If the array is multidimensional, you need to provide a list of lists, with an inner list for each dimension. If you want to extract three values from a two-dimensional array, you would provide a list containing two other lists. The first of these would contain the three row offsets, and the second would contain the three column offsets. You'll use this technique to easily sample pixel values at a list of locations in the next chapter. Try converting the random number array into two-dimensions and look at how it works:

```
>>> a = np.reshape(a, (3, 4))
>>> a
array([[16, 16, 18,  1],
       [14,  2, 18, 19],
       [ 2, 16, 10,  8]])
>>> a[[2, 0, 0], [0, 0, 3]]
array([ 2, 16,  1])
```

You can also use an array of Boolean values that's the same size as your data array, and the returned array will contain only the values that correspond to `True`. Here's an example of this, using the same array `a`:

```
>>> b
array([[False, False,  True, False],
       [False,  True, False, False],
       [ True,  True,  True, False]], dtype=bool)
>>> a[b]
array([18,  2,  2, 16, 10])
```

Why is this useful? Say you wanted to get the mean pixel value, but only for pixels that had a value greater than five. Using `where` to select the values of interest wouldn't work, because you'd still have to set the nonmatching ones to some value, which would mess up your mean calculation. Using Boolean indexing solves your problem.

```
>>> np.mean(a[a>5])
15.0
```

Sometimes you need to create a new array from scratch. If the cells need to be initialized to a certain value, you can use the `zeros` or `ones` functions. These return floating-point arrays by default, but you can specify the data type if needed. If you need a different number, you can create an array of ones and multiply that by the number you need:

```
>>> np.zeros((3,2))
array([[ 0.,  0.],
       [ 0.,  0.],
       [ 0.,  0.]])
>>> np.ones((2,3), np.int)
array([[1, 1, 1],
       [1, 1, 1]])
>>> np.ones((2,3), np.int) * 5
array([[5, 5, 5],
       [5, 5, 5]])
```

You might have noticed that `np.int` was provided as a second parameter to the `ones` examples. Arrays are created as floating-point by default, but you can specify a different data type if you need. This example didn't specify if it should be a 32-bit or 64-bit integer, and the result is system-dependent. To ensure that you get a 64-bit integer, use `np.int64`. A list of the available NumPy data types can be found at http://docs.scipy.org/doc/numpy/user/basics.types.html.

TIP NumPy data types and GDAL data types aren't the same thing, and you can't use them interchangeably.

If you need an empty array that doesn't have to be initialized to a certain value, you can use the `empty` function. This is faster than initializing an array, but be sure to fill all cells with real data eventually, because ones that you don't fill will contain garbage, like that shown here:

```
>>> np.empty((2,2))
array([[  2.50516998e-315,   2.50377043e-315],
       [  1.53313748e-316,   0.00000000e+000]])
```

You'll see examples of several more NumPy functions and techniques for working with arrays as you read through this chapter.

11.2 Map algebra

Map algebra is way of manipulating raster datasets using algebraic operations that you're already familiar with, such as addition and subtraction. In this case, however, two or more rasters are used instead of numbers. You can use these techniques to process raster data in many different ways, from simple to complex. You could touch up a dataset to make it look better on a map, or you could create entirely new datasets derived from one or more others.

Four main types of map algebra exist, all useful for different types of analyses. Several of the array examples in the previous section showed local analysis, where each operation works on individual pixels. This is what happens when you add two arrays together. Focal analyses use a few surrounding pixels, zonal operations work on pixels with the same value, and global analyses work on the entire array. We'll look at examples of all of these.

Before diving into the details, though, let's write a function that will save typing later, as shown in listing 11.1. It will create an output GeoTIFF with the same dimensions, geotransform, and projection as an existing dataset. The function will require five parameters: the existing dataset, the filename for the new dataset, a NumPy array containing data to write to the new image, an output data type, and an optional `NoData` value.

Listing 11.1 Function to save a new raster

```
def make_raster(in_ds, fn, data, data_type, nodata=None):
    """Create a one-band GeoTIFF.

    in_ds     - datasource to copy projection and geotransform from
    fn        - path to the file to create
    data      - NumPy array containing data to write
    data_type - output data type
    nodata    - optional NoData value
    """
    driver = gdal.GetDriverByName('GTiff')
    out_ds = driver.Create(
        fn, in_ds.RasterXSize, in_ds.RasterYSize, 1, data_type)
    out_ds.SetProjection(in_ds.GetProjection())
    out_ds.SetGeoTransform(in_ds.GetGeoTransform())
    out_band = out_ds.GetRasterBand(1)
    if nodata is not None:
        out_band.SetNoDataValue(nodata)
    out_band.WriteArray(data)
    out_band.FlushCache()
    out_band.ComputeStatistics(False)
    return out_ds
```

This code has nothing new. All it does is create a new raster using information from the existing dataset and the provided data type, write the data into this new raster, and compute statistics. It then returns the new dataset. All of this code would need to be in

the rest of the chapter listings, but this function will reduce it to one line. For the sake of convenience, it's already in the ospybook module.

11.2.1 Local analyses

Local map algebraic operations are probably the simplest to both understand and perform. They work on two or more arrays that are the same size, and an algebraic equation is applied to each set of pixel locations. Figure 11.1 shows an example of a local calculation that adds two 2D arrays together. This is a simple example, but the operations can be much more involved if required.

Adding two rasters together may not seem useful at first, but it can be. For example, I remember helping with a project many years ago that used this technique to rank land for conservation efforts. A few input rasters were created that ranked locations by individual variables, such as distance to riparian areas. Areas within a certain distance to water had the highest rank, and other distance intervals had different ranks. Another input raster ranked areas on biodiversity, and another on distance to existing developments. There were six or seven of these datasets, all with a small number of rank categories. They were added together to find the locations with the highest overall rank, and therefore the most important for conservation efforts. This simple model was then turned into an interactive online tool that allowed people to change their rankings of the different variables. If a user selected a different ranking structure for a variable, a new raster to reflect those priorities was created and the appropriate rasters were added together to get a new overall importance map. This gave planners a simple tool for exploring different scenarios without knowing anything about GIS. I'm sure much more sophisticated models exist online today, but this was in the days before online mapping was common.

Local analysis can be used for plenty of other things as well. Another common task using multispectral imagery is to compute various indices for tasks such as distinguishing between burned and unburned land or measuring nitrogen contained in vegetation. Let's look at an index used to measure "greenness," the normalized difference vegetation index (NDVI). The NDVI is a simple index that uses red and near-infrared wavelengths to produce a number that ranges from -1 to 1. Growing plants use red wavelengths for photosynthesis, but reflect near-infrared radiation, so a high ratio of these two measurements can indicate photosynthetic activity and healthy vegetation.

3	5	6	4
4	5	8	9
2	2	5	7
5	7	9	8

+

6	9	8	6
4	5	5	7
3	6	2	4
9	7	8	6

=

9	14	14	10
8	10	13	16
5	8	7	11
14	14	17	14

Figure 11.1 Local map algebra calculations work on a pixel-by-pixel basis, so the equation applies to pixels that fall in the same spatial location.

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.

Remember the near-infrared color composite from chapter 9 that clearly showed that stadium turf was artificial? Let's revisit that briefly in figure 11.2. Here you can see the natural color, red band, near-infrared band, near-infrared composite, and NDVI images. The red, near-infrared, and NDVI images are of single bands, where brighter areas have higher values. Notice that the vegetation is dark in the red band image but bright in the near-infrared one. Vegetation is absorbing red light and reflecting near-infrared, and these pixel values measure the amount being reflected back to the sensor, the same way our eyes see what's reflected back. The color infrared composite is a visual image only. Our eyes can see that vegetation appears red (unless you're reading a black-and-white copy of this book, in which case it's gray and doesn't look much different from the natural color image), but that's not useful if you want to use the data in an analysis. This is where the NDVI comes in. The practice fields in the NDVI image are bright, meaning they have high values and represent growing vegetation. The stadium field is dark, making it easy to determine that it's artificial.

In the following example, you'll calculate the NDVI that's shown in figure 11.2E. The formula is simple:

$$NDVI = \frac{NIR - RED}{NIR + RED}$$

You can use NumPy to apply this equation to two arrays, where one holds the red values and the other the near-infrared. You already know how to read data into NumPy arrays, because you did it in chapter 9. One potential problem exists, however. It's possible for both the red and near-infrared pixels to have 0 values, in which case the denominator is also 0, and we all know that you can't divide by 0. This situation probably doesn't exist in the example data we're using, but if you're using a satellite image you may have a large number of 0 values around the edges, and this becomes important.

You have several ways of dealing with this problem, and you might get different advice depending on who you talk to. The first is to proceed as if there were no problem, although I don't suggest this approach. By default, NumPy will warn you that it ran into errors, but the rest of the calculations will be fine (you can change this behavior, however, so if your settings are different, then things might crash). However, the output will have invalid numbers for the pixels that couldn't be calculated, so you'll have to deal with that somehow. If both the numerator and denominator of the equation are equal to 0, then the output is the `np.nan` value, which stands for *not a number*. If only the denominator is 0, then the output is set to the `np.inf` value, which stands for *infinity*. If you leave these values in your dataset, not only will it affect your statistics calculations, but different software will treat the values differently. For these reasons,

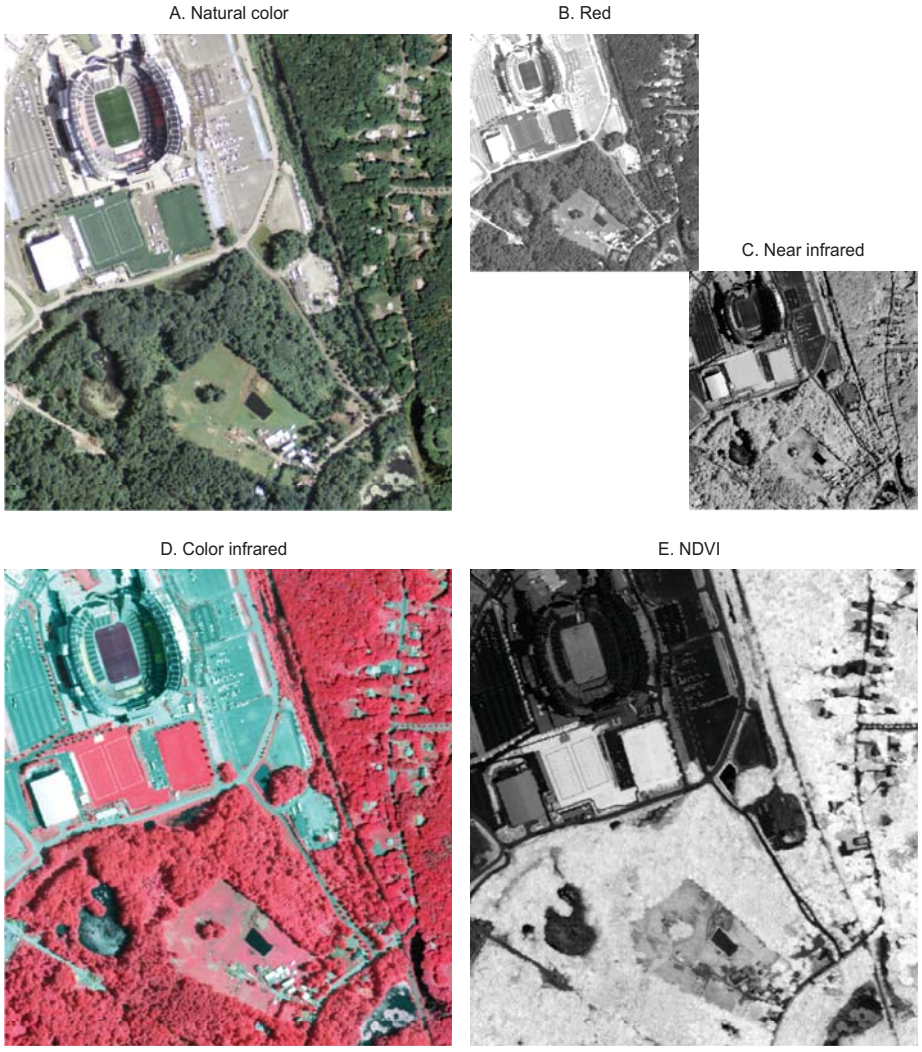


Figure 11.2 Different examples of looking at the same image in different ways. Image A is the natural color image composed of the red, green, and blue wavelengths, and B and C are single bands, where bright areas have higher pixel values. Image D is a visual representation that allows people to see what's vegetation and what isn't, but doesn't help with data analysis. Image E, however, is a single NDVI band in which higher values represent growing vegetation. Not all differences are readily apparent in black and white, but a color version of this figure is available online.

you'll probably want to set the invalid pixels to `NoData` so that things are standardized. You could do that by checking for pixels equal to either of these values and replacing them with another number, like this:

```
ndvi = (nir - red) / (nir + red)
ndvi = np.where(np.isnan(ndvi), -99, ndvi)
ndvi = np.where(np.isinf(ndvi), -99, ndvi)
```

Then you would also set the `NoData` value for your output band to whatever number you used, in this case -99. I like to use -99 because I know it isn't a valid number for my use cases and it's easy for me to remember, but software packages tend to use much larger numbers.

You might think you could deal with the problem by doing the calculation on only the pixels that don't have 0 as a denominator, like this:

```
ndvi = np.where(nir + red > 0, (nir - red) / (nir + red), -99)
```

This will run faster than the first example, but you'll still get division errors and risk a crash. At least you don't have to check for `nan` or `inf`, because everywhere with a division by 0 will be assigned -99 during the calculation.

TIP Change NumPy's behavior when it encounters floating-point errors with the `numpy.seterr` function.

A better solution to the division by 0 problem is to use masked arrays, which allow you to completely ignore certain pixels during the calculation. This will get rid of the division errors, and also makes it explicit what pixels you're ignoring. The idea is that you mask out the pixels that you want to ignore, do your calculations, and then fill in the missing pixels with your `NoData` value. Check out the following listing for an example of this in action.

Listing 11.2 Compute NDVI for a NAIP image

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

os.chdir(r'D:\osgeopy-data\Massachusetts')
in_fn = 'm_4207162_ne_19_1_20140718_20140923_clip.tif'
out_fn = 'ndvi.tif'

ds = gdal.Open(in_fn)
red = ds.GetRasterBand(1).ReadAsArray().astype(np.float)
nir = ds.GetRasterBand(4).ReadAsArray()
red = np.ma.masked_where(nir + red == 0, red)
ndvi = (nir - red) / (nir + red)
ndvi = ndvi.filled(-99)

out_ds = pb.make_raster(
    ds, out_fn, ndvi, gdal.GDT_Float32, -99)
overviews = pb.compute_overview_levels(out_ds.GetRasterBand(1))
out_ds.BuildOverviews('average', overviews)
del ds, out_ds
```

Mask the red band

Fill the empty cells

Set NoData to the fill value

This example uses the same input image as that shown in figure 11.2. This dataset is from the National Agriculture Imagery Program (NAIP), part of the United States Department of Agriculture. These aerial images are acquired periodically, with different states processed in different years. Although the visible red, green, and blue bands

are always collected so that the images are natural color, sometimes a fourth, near-infrared band is also collected. That's the case with the image used here. The first band is red light, and the fourth is near-infrared, which is why you read these two bands in at the beginning. Because the output is floating-point, you want to ensure that floating point math is used, so you convert the red band from byte to floating-point as you read it into the NumPy array.

Once you have the data in memory, you mask out the red array in all locations where the sum of the two arrays is 0. Although you could also mask the near-infrared data, you don't need to because if one array has a masked value, then no computations happen on that pixel, so it doesn't matter what value the other input arrays have. If you'd rather create a separate mask because you want to apply it to multiple arrays, you could do something like this instead:

```
mask = np.ma.equal(nir + red, 0)
red = np.ma.masked_array(red, mask)
```

Once you mask out the bad pixels, you apply the NDVI equation to the two arrays to create a third one with valid NDVI values in most pixels, but with bad pixels masked out and containing no value. You want your output image to have `NoData` in those locations, so you fill those pixels in with -99 and then make sure to set -99 as the band's `NoData` value later. All that's left to do is save your new NDVI array to a new dataset with the same projection and geotransform as the original NAIP image.

11.2.2 Focal analyses

Focal analyses use the pixels that surround the target pixel in order to calculate a value. For a given cell in the output, the value is calculated based on the corresponding cell and its neighbors in the input dataset. This is also called a *moving window* analysis because you can think of it as "window" of cells centered on each pixel in turn. Once the value for the target pixel is calculated, the window moves to the next pixel. Figure 11.3 shows how a 3 x 3 window would "move" across an image. The output values of the dark pixels are calculated using the nine surrounding lightly-shaded

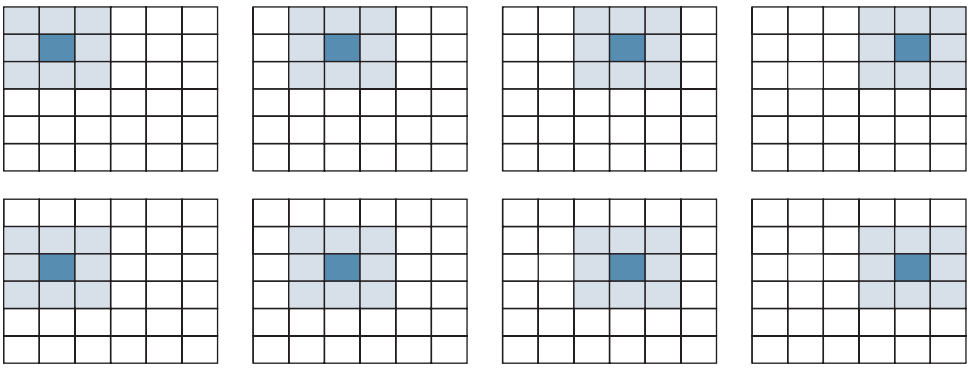


Figure 11.3 In a 3 x 3 moving window analysis, the output value for each dark pixel is calculated using the nine surrounding lightly-shaded input pixels.

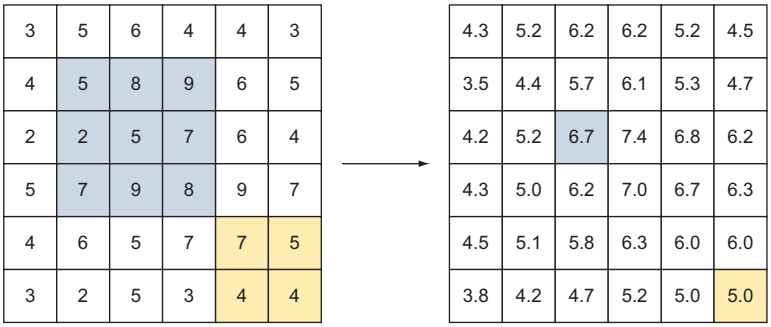


Figure 11.4 A 3 x 3 moving window that calculates the average value of the nine surrounding pixels (or less if the target pixel is on the edge). The shaded areas correspond to the window of input pixels that produce one output pixel value.

input pixels. These types of operations are common for smoothing data and removing random noise. In fact, you've probably used similar filters to touch up your own digital photos. Focal analyses can also be used for anything else that requires input from surrounding pixels, such as computing slope and aspect for an elevation dataset.

Figure 11.4 shows an example of a smoothing filter that computes the mean of a 3 x 3 moving window. The value of each output pixel is the average value of the nine surrounding pixels in the input. The exception to this is if the target pixel is on the edge, so that there aren't a full nine surrounding pixels. In this particular example, the average of the available pixels is used, but you have many ways of dealing with the edge problem. In the figure, the shaded regions in the input (left) raster show the cells that are used to compute the output value for the corresponding shaded cells in the raster on the right.

If the input data array from figure 11.4 is called `indata` and the result is called `outdata`, then the upper shaded output pixel in the figure is computed like this:

```
outdata[2,2] = (indata[1,1] + indata[1,2] + indata[1,3] +
                indata[2,1] + indata[2,2] + indata[2,3] +
                indata[3,1] + indata[3,2] + indata[3,3]) / 9
```

This is the average of the nine surrounding pixels. Thankfully, you have a shorter way to write the same thing:

```
outdata[2,2] = np.mean(indata[1:4, 1:4])
```

Using this information, you might be tempted to loop through the rows and columns of a raster to implement a moving window like this one, especially if you have background in a language such as C. To simplify and eliminate special cases where you don't have nine input pixels, you might throw out the outer rows and columns, and then your code would be similar to this:

```
outdata = np.zeros(indata.shape, np.float32)
for i in range(1, rows-1):
    for j in range(1, cols-1):
        outdata[i,j] = np.mean(indata[i-1:i+2, j-1:j+2])
```

Don't try this at home!

This example would run, but it would be excruciatingly slow, and unless your raster was small, you'd wait a long time for the output. It's a bad idea to implement looping like this on a NumPy array if you don't absolutely have to. You're much better off using array slices, and then your processing speed will be closer to the speed you could get with C. To do this, you need to create nine slices, each one corresponding to one of the nine input pixels, as shown in figures 11.5 and 11.6. The first figure shows a small raster with six rows and columns, and the lightly shaded cells correspond to the slice specified with the text below each example. The dark outline defines the 3 x 3 window around the cell at index [2, 2].

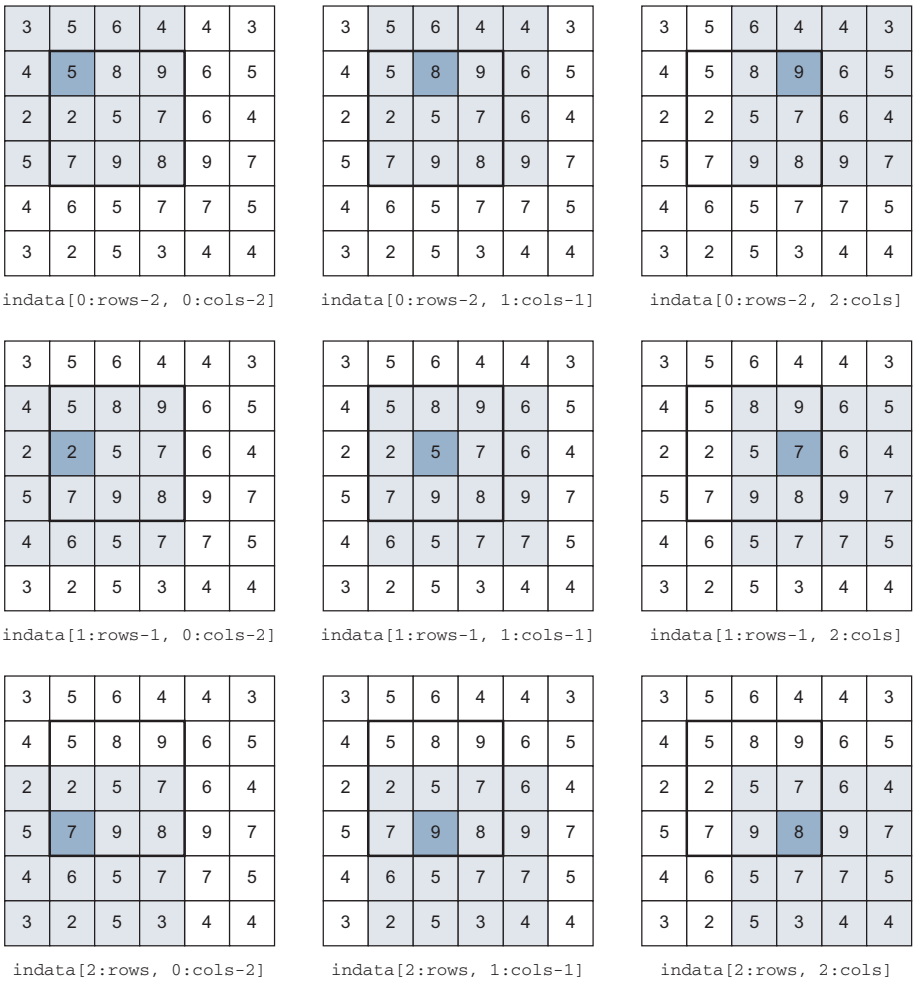


Figure 11.5 The slices that are used in a 3 x 3 moving window. Each example shows the same input data, but the lightly shaded cells are the slices defined by the text below the example. The dark outline defines the window around pixel [2, 2]. The darker shaded cells are all at index [1, 1] inside the corresponding slice.

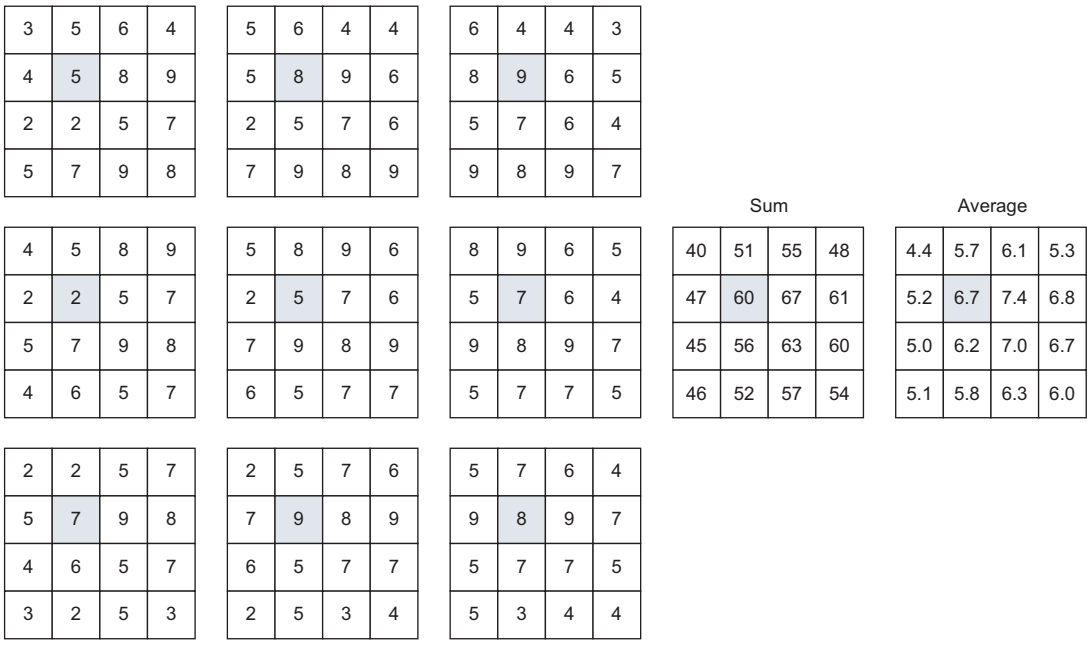


Figure 11.6 The individual slices created in figure 11.5, along with their sum and average. The shaded cells at index [1, 1] in each slice are the same cells as those in the outlined window in figure 11.5, so averaging the slices is equivalent to averaging the cells in the window.

Figure 11.6 shows these same slices with the cell at index [1, 1] highlighted. Compare the values of these highlighted pixels to the values of the pixels inside the dark outline in figure 11.5. They're the same, so if you take the average of the slices, the value at index [1, 1] will be the average of the nine pixels outlined in figure 11.5. In fact, the output contains the average of all complete 3 x 3 windows in the original dataset. Again, this leaves the edge rows and columns out of the resulting data, for the sake of simplicity. You'd need to cut off more rows and columns from the edges for larger moving windows. For example, a 5 x 5 window would cut off two on each side instead of one.

You could create all nine slices, add them together, and then divide by 9, like this:

```
outdata = np.zeros(indata.shape, np.float32)
outdata[1:rows-1, 1:cols-1] = (
    indata[0:-2, 0:-2] + indata[0:-2, 1:-1] + indata[0:-2, 2:] +
    indata[1:-1, 0:-2] + indata[1:-1, 1:-1] + indata[1:-1, 2:] +
    indata[2:  , 0:-2] + indata[2:  , 1:-1] + indata[2:  , 2:]) / 9
```

That looks like a pain, but again, I have an easier way to do it. If the slices are all stacked into a three-dimensional array, then you can use the `mean` function, which would definitely be simpler. The `dstack` function will stack the slices on top of each other, which is what you need. But you still need to get all of the slices so you can pass them to `dstack`. You could type everything out again, but that isn't any easier than before. Instead, you could use a loop to get each slice and add it to a list. To do this,

you need to loop through three rows and three columns. Assuming that you use loop indices 0-2, the current index can be used as the row or column to start the slice. You know that when a slice starts at row 0, then it needs to end at 2 less than the number of rows. If you add 1 to the starting index, then you need to add 1 to the ending index as well. Therefore, you can find the ending index by adding the starting index to the number rows minus 2:

```
slices = []
for i in range(3):
    for j in range(3):
        slices.append(indata[i:rows-2+i, j:cols-2+j])
```

Not only does this require less typing, but it scales much easier to larger windows, as you'll see in a bit. But now that you have a list of slices, you can stack them in the third dimension with the `dstack` function, which returns a three-dimensional array that can be used to compute means:

```
stacked = np.dstack(slices)
outdata = np.zeros(indata.shape, np.float32)
outdata[1:-1, 1:-1] = np.mean(stacked, 2)
```

By default, the `mean` function returns the mean of all pixels in the array, but you want the mean calculated for each set of pixels in a single spatial location, like a local analysis. To do this, tell the function which axis you'd like the mean calculated on. The third dimension is axis 2, so if you specify that, you'll get an array with the same numbers of rows and columns as the stacked array, with the value of each cell being the mean of the nine slices in that location. The slices have two less rows and columns than the original data set, however, so you create a zero-filled array the same size as the original data, and then insert the array containing the means into the middle of it, cutting off a row and column on each side.

This method of getting the nine slices can be easily generalized into a function that will return slices of any size you want (well, as long as the dimensions are odd numbers—even numbers don't work well because there's no middle cell). This function, which is in the ospybook module, is shown in the next listing.

Listing 11.3 Function to get slices of any size from an array

```
def make_slices(data, win_size):
    """Return a list of slices given a window size.

    data     - two-dimensional array to get slices from
    win_size - tuple of (rows, columns) for the moving window
    """
    rows = data.shape[0] - win_size[0] + 1
    cols = data.shape[1] - win_size[1] + 1
    slices = []
    for i in range(win_size[0]):
        for j in range(win_size[1]):
            slices.append(data[i:rows+i, j:cols+j])
    return slices
```

Calculate slice size

Create the slices

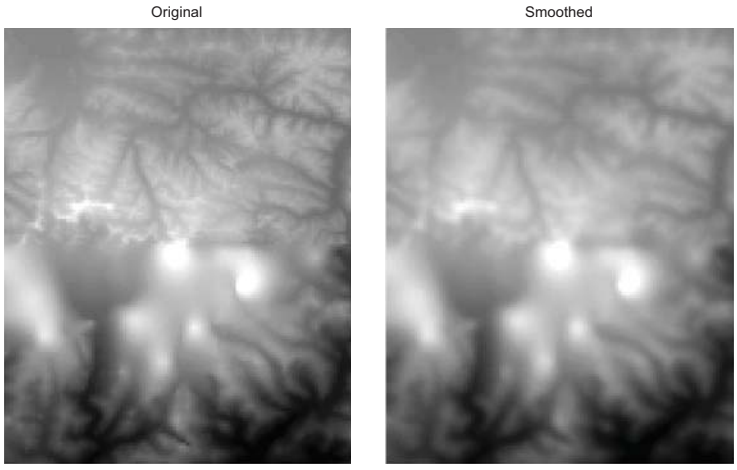


Figure 11.7 A digital elevation model of the area surrounding Mt. Everest, along with a version that has been smoothed using a 3 x 3 moving average filter

Now you can use everything you've learned so far to run an average smoothing filter on an elevation dataset. Figure 11.7 shows a DEM of the area surrounding Mt. Everest. For some reason, a seamline runs right through the middle, and the northern half looks better than the southern half. I thought that perhaps smoothing the dataset would make the seamline less obvious. Whether it did or not is open to debate, but the smoothed image does look different from the original. This is especially obvious in the northern part of the image, where the contours are less distinct in the smoothed version. The following listing shows the code to apply the filter.

Listing 11.4 Smooth an elevation dataset

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

in_fn = r"D:\osgeopy-data\Nepal\everest.tif"
out_fn = r'D:\Temp\everest_smoothed_edges.tif'
in_ds = gdal.Open(in_fn)
in_band = in_ds.GetRasterBand(1)
in_data = in_band.ReadAsArray()
slices = pb.make_slices(in_data, (3, 3))
stacked_data = np.ma.dstack(slices)

rows, cols = in_band.YSize, in_band.XSize
out_data = np.ones((rows, cols), np.int32) * -99
out_data[1:-1, 1:-1] = np.mean(stacked_data, 2)

pb.make_raster(in_ds, out_fn, out_data, gdal.GDT_Int32, -99)
del in_ds
```

Stack the slices

Put result in middle of output

Initialize output to NoData

Although it took a while to work up to it, the filtering code turned out to be simple. You use your `make_slices` function to create the nine slices, stack them into a

three-dimensional array, and then use the `mean` function to calculate the average across the third dimension. Because the slices are smaller than the original data, you put the result into the middle of an array of the correct size that has already been initialized to the `NoData` value. This ensures that the ignored edges are set to `NoData`, as long as you remember to pass that value to the `make_raster` function.

Nothing is stopping you from applying much more complicated functions to the cells that make up the moving window. In fact, this is exactly what you'd want to do for many analyses. One example is computing slope from an elevation model. Several algorithms calculate slope, and one of them is shown in figure 11.8.

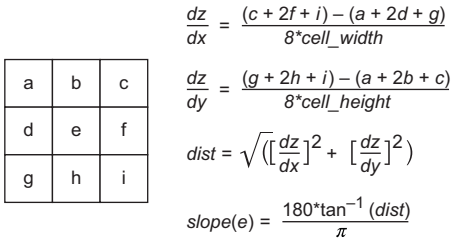


Figure 11.8 The algorithm for computing the slope of cell e from elevation values in the surrounding cells

The next listing shows code for calculating the slope of the Mt. Everest DEM using these equations. Note that for this algorithm to work properly, the elevation units must be the same as the horizontal ones. For example, if your dataset uses a UTM projection, then the coordinates are expressed in meters, so the elevation values must also be meters.

Listing 11.5 Compute slope from DEM

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

in_fn = r"D:\osgeopy-data\Nepal\everest_utm.tif"
out_fn = r'D:\Temp\everest_slope.tif'

in_ds = gdal.Open(in_fn)
cell_width = in_ds.GetGeoTransform()[1]
cell_height = in_ds.GetGeoTransform()[5]
band = in_ds.GetRasterBand(1)
in_data = band.ReadAsArray().astype(np.float)
out_data = np.ones((band.YSize, band.XSize)) * -99

slices = pb.make_slices(in_data, (3, 3))
rise = ((slices[6] + (2 * slices[7]) + slices[8]) -
        (slices[0] + (2 * slices[1]) + slices[2])) / \
       (8 * cell_height)
run =  ((slices[2] + (2 * slices[5]) + slices[8]) -
        (slices[0] + (2 * slices[3]) + slices[6])) / \
       (8 * cell_width)

dist = np.sqrt(np.square(rise) + np.square(run))
out_data[1:-1, 1:-1] = np.arctan(dist) * 180 / np.pi

pb.make_raster(in_ds, out_fn, out_data, gdal.GDT_Float32, -99)
del in_ds
```

Initialize output array with -99

Output edges don't get slope data

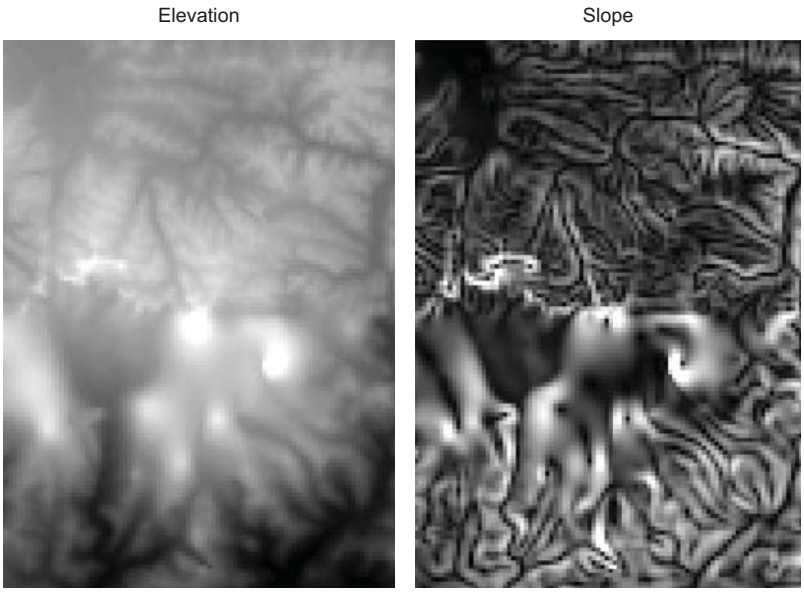


Figure 11.9 The original Mt. Everest DEM and a slope raster derived from it

This is more complicated than the smoothing example, but it's still not too bad. Most of it is made up of calculating the slope. You do need to know the order that the slices are stored in the `slices` list to use the correct one in each part of the equation, however. The `make_slices` function returns them in the same order as if you were reading left to right and down, or in other words, in alphabetical order if you refer to figure 11.8. Unlike the smoothing example, in this case you don't stack the slices into a three-dimensional array because you need to reference them individually in the slope equations. Again, you make sure that the edges are set to the `NoData` value. The output looks like that shown in figure 11.9.

USING SCIPY FOR FOCAL ANALYSIS

SciPy is a versatile Python module designed for scientific data analysis, and it uses NumPy arrays to store large amounts of data. It has submodules for interpolation, Fourier transforms, linear algebra, statistics, signal processing, and image processing, among others. The multidimensional image processing submodule contains filtering functions that can be used to perform the same operations you did with NumPy. It's probably easier to use SciPy than NumPy, but hopefully now you understand enough about working with NumPy arrays that you can figure out how to solve other problems that you might run into.

One advantage to using SciPy is that it will handle the edge problems for you by filling in extra cells around the edges so that the calculations can be performed on all cells. It has several different ways of populating these extra pixels, and you can decide which one you want to use. The default is the "reflect" mode, which repeats the values

near the edge, but in the opposite order. You can also use the nearest value, a constant value of your own choosing, or a few other value-repeating methods.

One of the built-in filters in SciPy is a uniform filter, which is a smoothing filter that works the same as the smoothing filter from listing 11.4. This next listing shows how you use it.

Listing 11.6 Smoothing filter using SciPy

```
import os
import scipy.ndimage
from osgeo import gdal
import ospybook as pb

in_fn = r"D:\osgeopy-data\Nepal\everest.tif"
out_fn = r'D:\Temp\everest_smoothed.tif'

in_ds = gdal.Open(in_fn)
in_data = in_ds.GetRasterBand(1).ReadAsArray()

out_data = scipy.ndimage.filters.uniform_filter(
    in_data, size=3, mode='nearest')

pb.make_raster(in_ds, out_fn, out_data, gdal.GDT_Int32)
del in_ds
```

Run the filter

As you can see, running the actual filter only requires one line of code, and the rest of it is dealing with reading and writing the data. The only required argument to `uniform_filter` is the NumPy array containing the data to smooth, but you have several optional parameters. You use two of them here. The `size` parameter specifies the size of the moving window to use, and you don't really need to use it here because the default value is 3 anyway. You also use the `mode` parameter to change the method of dealing with the edges so that the closest pixel values are used to fill in the edges.

Other built-in filters exist, including minimum, maximum, and median values. But what about more complicated situations such as the slope calculation? All you need to do is create a function that performs the calculation that you want and then pass that to a generic filter function, as shown in the following listing.

Listing 11.7 Calculate slope using SciPy

```
import os
import numpy as np
import scipy.ndimage
from osgeo import gdal
import ospybook as pb

in_fn = r"D:\osgeopy-data\Nepal\everest_utm.tif"
out_fn = r'D:\Temp\everest_slope_scipy2.tif'

def slope(data, cell_width, cell_height):
    """Calculates slope using a 3x3 window.
```

```
    data        - 1D array containing the 9 pixel values, starting
                  in the upper left and going left to right and down
    cell_width  - pixel width in the same units as the data
    cell_height - pixel height in the same units as the data
    """
    rise = ((data[6] + (2 * data[7]) + data[8]) -
            (data[0] + (2 * data[1]) + data[2])) / \
           (8 * cell_height)
    run =  ((data[2] + (2 * data[5]) + data[8]) -
            (data[0] + (2 * data[3]) + data[6])) / \
           (8 * cell_width)
    dist = np.sqrt(np.square(rise) + np.square(run))
    return np.arctan(dist) * 180 / np.pi

in_ds = gdal.Open(in_fn)
in_band = in_ds.GetRasterBand(1)
in_data = in_band.ReadAsArray().astype(np.float32)

cell_width = in_ds.GetGeoTransform()[1]
cell_height = in_ds.GetGeoTransform()[5]
out_data = scipy.ndimage.filters.generic_filter(
    in_data, slope, size=3, mode='nearest',
    extra_arguments=(cell_width, cell_height))

pb.make_raster(in_ds, out_fn, out_data, gdal.GDT_Float32)
del in_ds
```

Use floating-point

Run the filter

The first thing you do is write a custom filter function called `slope` that contains the exact same math as before, so it should look familiar. The first argument to your filter function must be a one-dimensional array of data that will be used for the calculation. Conveniently, the cell values will be in the same order that you used earlier with your `make_slices` function. The first value is the upper-left pixel, the second is the upper-middle pixel, and so on, until ending with the lower-right pixel. If you need it to, your function can also take additional parameters, but this isn't a requirement for custom filters. In this case, you need to know the pixel dimensions for the slope calculation, so your function also requires pixel width and height.

Once you have your custom filter function, you provide it as a parameter when calling the SciPy `generic_filter` function. The first argument to `generic_filter` is the NumPy array containing data to filter and the second is the filter function to use. These are the only required parameters, but once again, you can use optional ones. In this case, you specify a 3 x 3 moving window, but it's possible to use different sizes, or even use a Boolean array to indicate exactly which cell values to pass to your filter. You can read more about this in the SciPy documentation. Once again, you change the default method of dealing with the array edges, and finally, you provide a tuple containing the extra arguments that your function requires. The `generic_filter` function passes the appropriate pixel values to your function to calculate each output cell value.

BREAKING UP FOCAL ANALYSES

What if you want to do a moving window analysis but don't have enough RAM to hold everything in memory? You can break the image up into chunks, but instead of

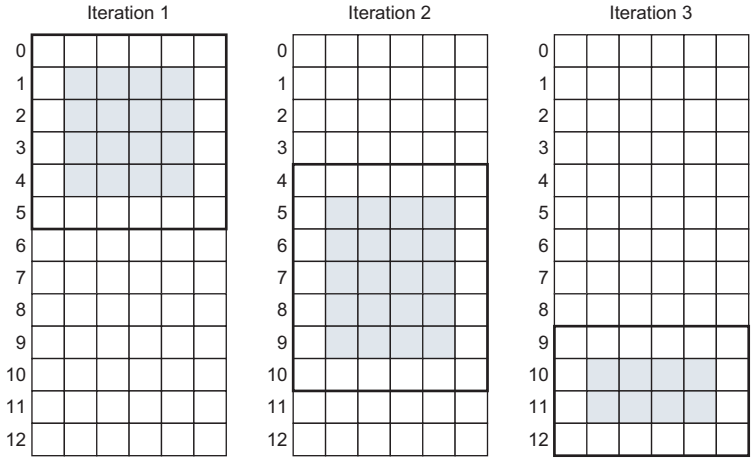


Figure 11.10 Breaking up an image into overlapping chunks. The thick outlines show the cells read from disk, and the shaded cells are the ones that get valid data and are written to the output.

processing discrete sets of data, have them overlap each other. Figure 11.10 shows an example of reading in multiple rows at a time using a step parameter of 5. More than five rows get read in each time, though, because of the overlap. The dark outlines show the rows that are processed, and the shaded areas are the cells that get valid data in that iteration. Because this is a 3 x 3 window, it has one empty row and column on each side. The idea is to tack one row on the top and one on the bottom of each chunk so that every row ends up with valid data. The first time through, one extra row is added at the bottom so six rows get processed instead of five. The second time through, two extra rows are processed. To get an extra row at the top, the starting offset is moved to an earlier row. For example, with a step value of 5, the second iteration would normally start reading at row 5, but instead it starts at row 4 in this example. The third time through is similar, except that the available rows are limited so only four are read in. You can see that all of the rows except the top and bottom end up with valid data.

Although the Everest dataset is small, pretend for a moment that it's too large to process all at once, but you have enough RAM to process approximately 100 rows at a time. The following listing shows how you could do this.

Listing 11.8 Focal analysis broken into chunks

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

in_fn = r"D:\osgeopy-data\Nepal\everest.tif"
out_fn = r'D:\Temp\everest_smoothed_chunks.tif'

in_ds = gdal.Open(in_fn)
in_band = in_ds.GetRasterBand(1)
```

```
xsize = in_band.XSize
ysize = in_band.YSize

driver = gdal.GetDriverByName('GTiff')
out_ds = driver.Create(out_fn, xsize, ysize, 1, gdal.GDT_Int32)
out_ds.SetProjection(in_ds.GetProjection())
out_ds.SetGeoTransform(in_ds.GetGeoTransform())
out_band = out_ds.GetRasterBand(1)
out_band.SetNoDataValue(-99)

n = 100
for i in range(0, ysize, n):
    if i + n + 1 < ysize:
        rows = n + 2
    else:
        rows = ysize - i
    yoff = max(0, i - 1)

    in_data = in_band.ReadAsArray(0, yoff, xsize, rows)
    slices = pb.make_slices(in_data, (3, 3))
    stacked_data = np.ma.dstack(slices)
    out_data = np.ones(in_data.shape, np.int32) * -99
    out_data[1:-1, 1:-1] = np.mean(stacked_data, 2)

    if yoff == 0:
        out_band.WriteArray(out_data)
    else:
        out_band.WriteArray(out_data[1:], 0, yoff + 1)

out_band.FlushCache()
out_band.ComputeStatistics(False)
del out_ds, in_ds
```

Read two extra rows if possible

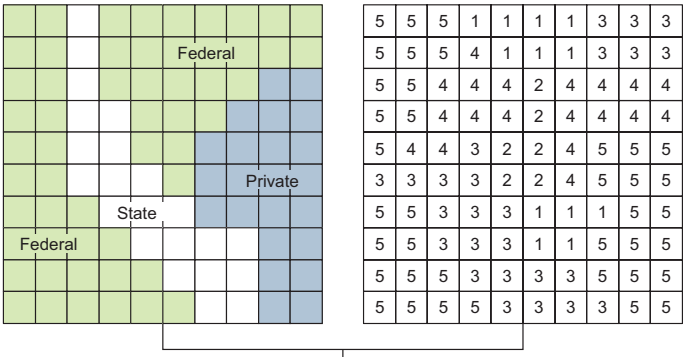
Don't start before row 0

Don't overwrite good data from previous chunk

Much of this resembles code you've written before. Remember that you're going to read in two extra rows if possible, so you need to take those extra rows into account when checking if enough rows exist to read an entire chunk. You also make sure you don't try to use -1 as a row offset in the first iteration. Once you determine how many rows to grab, you read them in and process them as before. For all but the first chunk, though, you make sure not to write the first row of the processed data to the output. If you had, it would've overwritten the good data written during the previous iteration. Because you're ignoring this first row, you also have to increase the row offset you use for writing.

11.2.3 Zonal analyses

Zonal analyses work on cells that share a certain value, or belong to the same zone. The zones are usually defined by one raster and the analysis performed using values from a second one. For example, if you have a raster showing land ownership categories such as federal, state, and private, and a second raster showing landcover, you could use the ownership categories as zones to determine the acreage of each landcover type within each ownership category, as shown in figure 11.11.



5	5	5	1	1	1	1	3	3	3
5	5	5	4	1	1	1	3	3	3
5	5	4	4	4	2	4	4	4	4
5	5	4	4	4	2	4	4	4	4
5	4	4	3	2	2	4	5	5	5
3	3	3	3	2	2	4	5	5	5
5	5	3	3	3	1	1	1	5	5
5	5	3	3	3	1	1	5	5	5
5	5	5	3	3	3	3	5	5	5
5	5	5	5	3	3	3	3	5	5

5	5	5	1	1	1	1	3	3	3
5	5	5	4	1	1	1	3	3	3
5	5	4	4	4	2	4	4	4	4
5	5	4	4	4	2	4	4	4	4
5	4	4	3	2	2	4	5	5	5
3	3	3	3	2	2	4	5	5	5
5	5	3	3	3	1	1	1	5	5
5	5	3	3	3	1	1	5	5	5
5	5	5	3	3	3	3	5	5	5
5	5	5	5	3	3	3	3	5	5

	1	2	3	4	5
State	3	1	10	4	4
Federal	7	5	15	8	20
Private	2	0	0	7	14

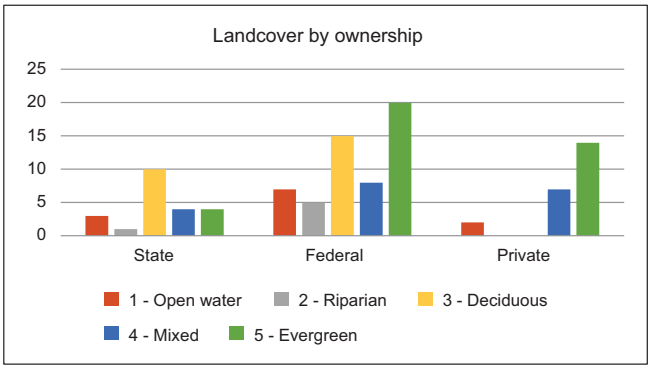


Figure 11.11 An example of a zonal analysis using ownership and landcover type. The number of pixels for each landcover are counted per ownership zone.

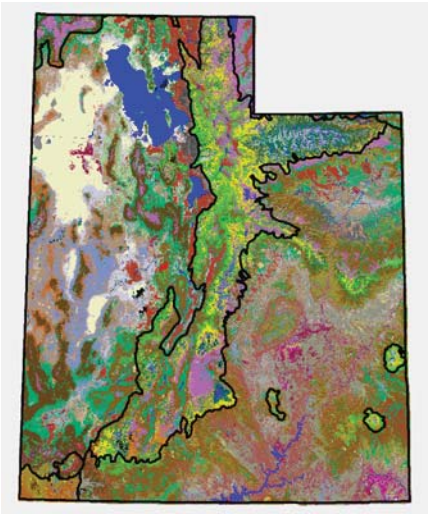
First let's look at how you could do this with NumPy and then a more flexible method using SciPy. What you want here is basically a two-dimensional histogram. A regular histogram gives you the number of items in each bin, but in this case, you want to treat zones as one set of bins and landcover as another set of bins, and then get the count for each combination of zone and landcover bin. There are several ways you can define

your bins, including letting NumPy do it for you, but here you'll see how to do it yourself. A set of bins is defined by an array of bin edges. The first number in the array is the lower bound for the first bin, the second number is the upper, non-inclusive bound for the first bin and also the lower, inclusive bound for the second bin, and so on. The last number in the array is the upper bound of the last bin. One easy way to get the bins for this particular use case is to get the unique values in the dataset. The NumPy `unique` function returns these values in sorted order. Because the lower bound is inclusive, this list of numbers would create the lower bound for bins corresponding to the dataset values. All that's left is to add a larger number to the end to form the upper bound for the last bin. The following function creates this array of bin edges for you:

```
def get_bins(data):
    """Return bin edges for all unique values in data. """
    bins = np.unique(data)
    return np.append(bins[~np.isnan(bins)], max(bins) + 1)
```

Now that you know how to define bins, let's see how to use them with the NumPy `histogram2d` function to get the counts. The two required parameters for this function are the two arrays containing values to bin, and one of the optional arguments lets you specify the bins you want to use. If the values from the upper-left dataset shown in figure 11.11 are in an array called `zones`, and the values from the upper-right dataset are in an array called `landcover`, then you can get the two-way histogram like this:

```
>>> hist, zone_bins, landcover_bins = np.histogram2d(
...     zones.flatten(), landcover.flatten(),
...     [get_bins(zones), get_bins(landcover)])
>>> hist
array([[  3.,   1.,  10.,   4.,   4.],
       [  7.,   5.,  15.,   8.,  20.],
       [  2.,   0.,   0.,   7.,  14.]])
```



Notice a couple of things here. First, the `histogram2d` function wants the data arrays to be one-dimensional, and the `flatten` function takes care of that detail. The `histogram2d` function returns three values: a two-dimensional histogram and two sets of bins, one for each input array. The histogram rows correspond to the bins from the first array passed in, and the columns correspond to the second. In this case, the two bin outputs will be exactly what you pass in, but if you don't explicitly define your bins, then these two return values would tell you what bins were used for the calculation.

Figure 11.12 SWReGAP landcover classification with ecoregion boundaries drawn on top

If you have SciPy, you can accomplish this same thing with a more general SciPy function called `stats.binned_statistic_2d`. The following listing shows you how to use this to count up the number of pixels with each landcover class in each ecoregion zone shown in figure 11.12.

Listing 11.9 Zonal analysis with SciPy

```
import numpy as np
import scipy.stats
from osgeo import gdal

def get_bins(data):
    """Return bin edges for all unique values in data."""
    bins = np.unique(data)
    return np.append(bins, max(bins) + 1)

landcover_fn = r'D:\osgeopy-data\Utah\landcover60.tif'
ecoregion_fn = r'D:\osgeopy-data\Utah\utah_ecoIII60.tif'
out_fn = r'D:\Temp\histogram.csv'

eco_ds = gdal.Open(ecoregion_fn)
eco_band = eco_ds.GetRasterBand(1)
eco_data = eco_band.ReadAsArray().flatten()
eco_bins = get_bins(eco_data)

lc_ds = gdal.Open(landcover_fn)
lc_band = lc_ds.GetRasterBand(1)
lc_data = lc_band.ReadAsArray().flatten()
lc_bins = get_bins(lc_data)

hist, eco_bins2, lc_bins2, bn = \
    scipy.stats.binned_statistic_2d(
        eco_data, lc_data, lc_data, 'count',
        [eco_bins, lc_bins])
hist = np.insert(hist, 0, lc_bins[:-1], 0)
row_labels = np.insert(eco_bins[:-1], 0, 0)
hist = np.insert(hist, 0, row_labels, 1)
np.savetxt(out_fn, hist, fmt='%1.0f', delimiter=',')
```

Compute histogram

Add bin info to histogram

Save output

The first thing you do is read in the ecoregion and landcover datasets as flattened, one-dimensional arrays (because this function, like `histogram2d`, requires single-dimension arrays), and also calculate the required bins for each dataset. Then you create your histogram using the `binned_statistic_2d` function. The first two parameters are the same as the `histogram2d` function, namely, the two datasets that will be binned. Unlike `histogram2d`, this function can not only count occurrences but also calculate statistics, so it also requires a third array containing the values to calculate statistics on. Because you count the number of pixels, it doesn't matter if you use landcover or ecoregions in this case, but you use landcover. The next parameter to the function specifies which statistic you want to calculate, which is "count" in this case (other options are mean, median, sum, or a custom function that you provide). But

you could do something like calculate the mean elevation in each combination of ecoregion and landcover by passing an elevation dataset as the third parameter and "mean" as the fourth. Anyway, you provide the bin boundaries as the last argument, the way you did before. This returns the same outputs as the histogram function, along with one extra one that indicates which bin the data value fell into.

This time you also get ambitious and add several bin labels to your histogram. Because the different landcovers are the columns, you use those bins as the column labels. Remember that the last item in the bins array is the upper bound and isn't needed to label your bins. You insert all but the last number in the landcover bins array as the first row of the histogram. The `insert` function wants the data being inserting into, the index to insert at, the data to insert, and the axis, where 0 means a row. You use the same idea to insert the ecoregion bins as row labels, except that you have to add a placeholder for the first row, which is now landcover labels. You insert 0 at the beginning and then use axis 1 to insert a column at the beginning of your histogram.

If you wanted to know the area instead of pixel count, you could multiply the histogram array by the area of a pixel before adding the label row and column.

Once your table is complete, you write it out to a text file. The `fmt` parameter to `savetxt` specifies how you want the numbers formatted in the output. Without this, you'd probably get scientific notation; nothing is wrong with that, but here you specify integers instead. The `%` is the first character of a format string, the `1` means you want it to print out at least one digit, the `.0` means no numbers after the decimal point, and `f` means that it will be getting a floating-point number to work with. For more information on format strings, please see the NumPy documentation for `savetxt` at http://docs.scipy.org/doc/numpy/reference/generated/numpy.savetxt.html.

Your output should look something like figure 11.13.

	A	B	C	D	E	F	G
1	0	0	1	2	4	5	8
2	1	8	0	0	0	0	21819
3	2	0	0	0	0	969	0
4	3	0	0	2796	0	10505	84088
5	4	0	0	553	0	157349	0
6	5	0	0	0	0	0	101
7	6	0	5867	209925	49015	235654	3
8	7	0	0	13174	0	2314	0
9	255	12487181	0	0	0	2873	1285

Figure 11.13 The first few columns of the histogram output. The first column specifies the ecoregion (255 is `NoData` cells), and the top row specifies the landcover category. All other values are cell counts.

What if you wanted to know the most common landcover type in each ecoregion but didn't care about counts? In this case, you could use the one-dimensional `binned_statistic` function instead, because you only need to bin one ecoregion. Unfortunately, mode isn't one of the supported statistics types, but you can provide your own statistical function. All you need to do is write a simple function that returns the mode and then pass it to `binned_statistic`, like this:

```
def my_mode(data):
    return scipy.stats.mode(data)[0]

modes, bins, bn = scipy.stats.binned_statistic(
    eco_data, lc_data, my_mode, eco_bins)
```

You can use this technique to calculate whatever information you want, as long as it can be computed from a one-dimensional array.

11.2.4 Global analyses

Global functions work on the entire image, such as proximity analysis or cost distance. A *proximity analysis* determines the Euclidean distance of each cell to the nearest cell that's marked as a source, while *cost distance* determines the least cost of traveling between each cell and the nearest source, as determined by a cost surface. For example, if you're walking between two points in the mountains, the easiest, and therefore least costly path, may not be the shortest. In this case, the cost surface might be derived from elevation and slope rasters, and cells on a mountain pass would have a lower cost than cells on steep ridges.

GDAL has global analysis functions built in, and you're about to see how to use several of them to determine the distance to the nearest road for areas in the Frank Church—River of No Return Wilderness in Idaho (figure 11.14). You'll start off with a state-wide roads shapefile and a shapefile showing wilderness boundaries. Several polygons make up this wilderness area, so you'll select them out, get the extent of the selected polygons, and use that rectangle to select the roads you're interested in.

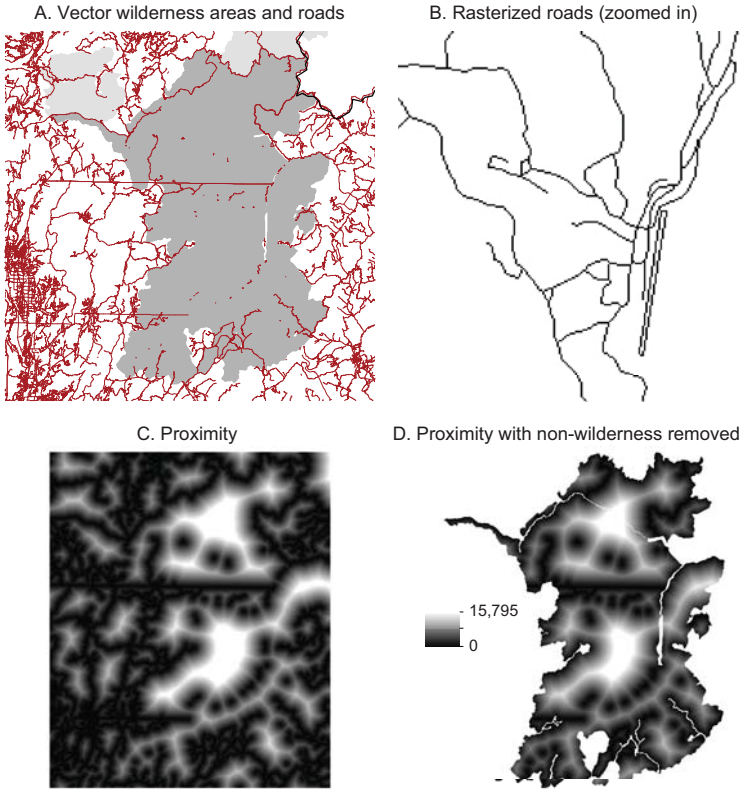


Figure 11.14 Data used for calculating average distance to roads within a wilderness area. A: The dark shaded area is the wilderness area of interest, and the lines are roads (the roads dataset has artifacts, but those will be treated as real data in the example). B: Roads are no longer smooth lines when rasterized. C: Results of the proximity analysis, where bright areas are further from roads. D: The proximity dataset with non-wilderness removed.

Then you can use GDAL to create a raster that has ones where there are roads and zeros everywhere else. This will be used as a source layer in order to determine distances from roads to every other pixel. The following listing shows how to accomplish all of this. The listing is long, but that's because the data haven't been preprocessed.

Listing 11.10 Proximity analysis

```
import os
import sys
from osgeo import gdal, ogr

folder = r'D:\osgeopy-data\Idaho'
roads_ln = 'allroads'
wilderness_ln = 'wilderness'
road_raster_fn = 'church_roads.tif'
proximity_fn = 'proximity.tif'
cellsize = 10

shp_ds = ogr.Open(folder)
wild_lyr = shp_ds.GetLayerByName(wilderness_ln)
wild_lyr.SetAttributeFilter(
    "WILD_NM = 'Frank Church - RONR'")
envelopes = \
    [row.geometry().GetEnvelope() for row in wild_lyr]
coords = list(zip(*envelopes))
minx, maxx = min(coords[0]), max(coords[1])
miny, maxy = min(coords[2]), max(coords[3])

road_lyr = shp_ds.GetLayerByName(roads_ln)
road_lyr.SetSpatialFilterRect(minx, miny, maxx, maxy)

os.chdir(folder)
tif_driver = gdal.GetDriverByName('GTiff')
cols = int((maxx - minx) / cellsize)
rows = int((maxy - miny) / cellsize)

road_ds = tif_driver.Create(road_raster_fn, cols, rows)
road_ds.SetProjection(
    road_lyr.GetSpatialRef().ExportToWkt())
road_ds.SetGeoTransform(
    (minx, cellsize, 0, maxy, 0, -cellsize))

gdal.RasterizeLayer(
    road_ds, [1], road_lyr, burn_values=[1],
    callback=gdal.TermProgress)

prox_ds = tif_driver.Create(
    proximity_fn, cols, rows, 1, gdal.GDT_Int32)
prox_ds.SetProjection(road_ds.GetProjection())
prox_ds.SetGeoTransform(road_ds.GetGeoTransform())
gdal.ComputeProximity(
    road_ds.GetRasterBand(1), prox_ds.GetRasterBand(1),
    ['DISTUNITS=GEO'], gdal.TermProgress)
```

Set the cell size for the analysis

Get extent of wilderness area

Select roads in wilderness extent

Create empty raster for roads

Burn roads into raster

Burn proximity data into new raster

```
wild_ds = gdal.GetDriverByName('MEM').Create(
    'tmp', cols, rows)
wild_ds.SetProjection(prox_ds.GetProjection())
wild_ds.SetGeoTransform(prox_ds.GetGeoTransform())
gdal.RasterizeLayer(
    wild_ds, [1], wild_lyr, burn_values=[1],
    callback=gdal.TermProgress)
```

Burn wilderness into temporary raster

```
wild_data = wild_ds.ReadAsArray()
prox_data = prox_ds.ReadAsArray()
prox_data[wild_data == 0] = -99
prox_ds.GetRasterBand(1).WriteArray(prox_data)
prox_ds.GetRasterBand(1).SetNoDataValue(-99)
prox_ds.FlushCache()
```

Set NoData outside wilderness

```
stats = prox_ds.GetRasterBand(1).ComputeStatistics(
    False, gdal.TermProgress)
print('Mean distance from roads is', stats[2])

del prox_ds, road_ds,  shp_ds
```

Because you're using statewide datasets, the first thing you do is find the extent of the wilderness area that you're interested in. To do this, you open the wilderness shapefile and set an attribute filter that selects out all records where the `WILD_NM` attribute value was equal to `'Frank Church - RONR'`. Because the layer's `GetExtent` function returns the extent of the entire layer, even when a filter has been applied, you have to come up with a different method to get bounding coordinates. The solution is to make a list of the extents for each of the selected polygons and then find the minimum and maximum coordinates from that. Creating the list of polygon extents is easy enough:

```
envelopes = [row.geometry().GetEnvelope() for row in wild_lyr]
```

Each tuple in this list contains the minimum and maximum x value, and the minimum and maximum y, in that order. Now if you zip these tuples together, you end up with four lists, one each for minimum and maximum x and y. From there, it's a simple matter of extracting the most extreme values in each list:

```
coords = list(zip(*envelopes))
minx, maxx = min(coords[0]), max(coords[1])
miny, maxy = min(coords[2]), max(coords[3])
```

Now that you have the bounding coordinates for the wilderness extent, you set a spatial filter on the roads layer to select only the roads that fall in that rectangle:

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

After getting your roads of interest, you turn them into a raster band that you use for your proximity analysis. The raster band must already exist, and then the vector features are burned into it. You figure out how many rows and columns will fit in the extent, given the cell size you chose early in the script. Smaller cell sizes result in more-precise distances, but they also greatly increase processing time. Ten meters is a reasonable size for this example.

```
cols = int((maxx - minx) / cellsize)
rows = int((maxy - miny) / cellsize)
```

Now you have all of the information necessary to create a new raster dataset, which you do. It needs a geotransform so that the roads can be burned into the correct locations, so you construct one from your bounding coordinates and numbers of rows and columns. You also copy the spatial reference from the roads layer. Remember that a layer's `GetSpatialRef` function returns a spatial reference object, but a raster dataset's `SetProjection` function requires a WKT string, which is why you have to get the layer's spatial reference as a string:

```
road_ds.SetProjection(road_lyr.GetSpatialRef().ExportToWkt())
road_ds.SetGeoTransform((minx, cellsize, 0, maxy, 0, -cellsize))
```

Now you can finally burn the roads into a raster band using the following function:

```
RasterizeLayer(dataset, bands, layer, [transformer], [transformArg],
               [burn_values], [options], [callback], [callback_data])
```

- `dataset` is the raster dataset containing the band(s) to burn into.
- `bands` is the list of bands to burn the data into, where the first one has index 1.
- `layer` is the OGR layer whose features will be burned into the raster bands.
- `transformer` is a GDAL transformer object to convert map coordinates into pixel offsets. If not provided, then the function will create its own using the geotransform.
- `transformArg` is the callback data for the transformer.
- `burn_values` is the list of values to burn into the raster wherever there are features. If this parameter is provided, it must be the same length as bands. The default for a byte array is 255.
- `options` is a list of key=value strings. See appendix E for a list of possibilities. (Appendixes C through E are available online on the Manning Publications website at https://www.manning.com/books/geoprocessing-with-python.)
- `callback` is a callback function for reporting burn progress.
- `callback_data` is data to be passed to the callback function.

You use this function to burn the value of 1 everywhere there was a road:

```
gdal.RasterizeLayer(
    road_ds, [1], road_lyr, burn_values=[1], callback=gdal.TermProgress)
```

If you load this raster into a GIS, it won't look like much until you start to zoom in. This is because the pixel size is so small that you'll need to zoom in to see many of them. When you do zoom in, you'll see that the roads are blocky, as in figure 11.15. This is a result of them now being represented as pixels instead of smooth vector lines.

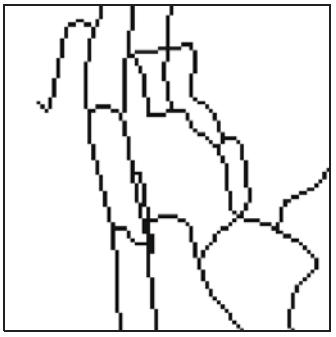


Figure 11.15 Rasterized roads

Once you have a raster representation of the roads, you're almost ready to compute proximity to them using the `ComputeProximity` function:

```
ComputeProximity(
    srcBand, proximityBand, [options], [callback], [callback_data])
```

- `srcBand` is the raster band containing the features to compute proximity to. By default, any non-zero pixels are considered features.
- `proximityBand` is the raster band to store the proximity measurements into.
- `options` is a list of key=value strings. See appendix E for a list of possibilities.
- `callback` is a callback function for reporting progress.
- `callback_data` is data to be passed to the callback function.

Like the `RasterizeLayer` function, the `ComputeProximity` function requires that the output raster band already exist. You create a new dataset and copy the spatial reference information and geotransform from your roads raster, and then calculate proximity using map distances instead of the default pixel distances:

```
gdal.ComputeProximity(
    road_ds.GetRasterBand(1), prox_ds.GetRasterBand(1),
    ['DISTUNITS=GEO'], gdal.TermProgress)
```

Although you now have a proximity raster, you only want statistics for the areas within the wilderness area. You could use a zonal analysis to calculate that information, or you could get rid of the non-wilderness data altogether. Either case requires rasterizing the wilderness polygons, though, so you do that in a similar manner as the roads, except you use the MEM driver to store the dataset in memory instead of writing it to disk. Then you read both the wilderness and proximity data into NumPy arrays and change all proximity values to -99 if the wilderness value is 0, which signifies that the pixel doesn't fall in a wilderness polygon.

```
prox_data[wild_data == 0] = -99
```

Once you save the data back to disk, you're able to correctly calculate statistics, and you also have a nice proximity dataset for use later. Figure 11.16 shows part of this dataset, zoomed in far enough to get an idea of how it works. The brighter the pixel, the longer the distance from a road.

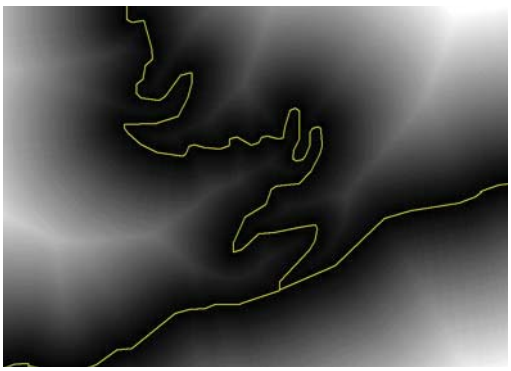


Figure 11.16 A small section of the proximity dataset with roads drawn on top. Brighter areas indicate longer distances from roads.

11.3 Resampling data

Back in chapter 9 you learned how to resample your data to different cell sizes by changing the size of the arrays used to hold the data. Other ways to

resample give you more control over the outcome, however. One simple approach is to use slices to keep pixel values at a specific interval and throw out everything in between. To do this, provide a step value when specifying your slice. A step value of 2 tells NumPy to keep every second value, 3 means keep every third value, and so on. This example shows you how to keep every other cell, reducing the rows and columns by half:

```
>>> data = np.reshape(np.arange(24), (4, 6))
>>> data
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11],
       [12, 13, 14, 15, 16, 17],
       [18, 19, 20, 21, 22, 23]])
>>> data[::2, ::2]
array([[ 0,  2,  4],
       [12, 14, 16]])
```

For this example, `data[0:4:2, 0:6:2]` provides the same results as `data[::2, ::2]`. Not providing starting and stopping indices around the first colon means that you want to start at the beginning and go to the end. The third number, after the second colon, is the step index. If you want to start at the second row and column instead of the first, you can do this:

```
>>> data[1::2, 1::2]
array([[ 7,  9, 11],
       [19, 21, 23]])
```

This should look familiar, because the results are similar to the automatic resampling that happens when you read data from a file into a differently sized array.

You can also increase the size of the array, which is how you'd decrease pixel sizes. To do this, use the NumPy `repeat` function, which wants an array of data, the number of times to repeat each value, and the axis to use. If you don't provide an axis, then the array is flattened to one dimension. An axis of 0 repeats the rows, and a value of 1 repeats columns, like this:

```
>>> np.repeat(data, 2, 1)
array([[ 0,  0,  1,  1,  2,  2,  3,  3,  4,  4,  5,  5],
       [ 6,  6,  7,  7,  8,  8,  9,  9, 10, 10, 11, 11],
       [12, 12, 13, 13, 14, 14, 15, 15, 16, 16, 17, 17],
       [18, 18, 19, 19, 20, 20, 21, 21, 22, 22, 23, 23]])
```

Notice how each column is repeated twice? To end up with each value repeated four times (so the rows and columns are both doubled), call `repeat` once on rows and once on columns:

```
>>> np.repeat(np.repeat(data, 2, 0), 2, 1)
array([[ 0,  0,  1,  1,  2,  2,  3,  3,  4,  4,  5,  5],
       [ 0,  0,  1,  1,  2,  2,  3,  3,  4,  4,  5,  5],
       [ 6,  6,  7,  7,  8,  8,  9,  9, 10, 10, 11, 11],
       [ 6,  6,  7,  7,  8,  8,  9,  9, 10, 10, 11, 11],
       [12, 12, 13, 13, 14, 14, 15, 15, 16, 16, 17, 17],
       [12, 12, 13, 13, 14, 14, 15, 15, 16, 16, 17, 17],
       [18, 18, 19, 19, 20, 20, 21, 21, 22, 22, 23, 23],
       [18, 18, 19, 19, 20, 20, 21, 21, 22, 22, 23, 23]])
```

But let's look at something more interesting. You can also use multiple slices to apply custom algorithms instead of using a single pixel value. For example, if you want to resample to pixels that are four times the original size (twice the length and twice the width), you could take the average of those four pixel values and use that as the new value. Figure 11.17 shows an example of this.

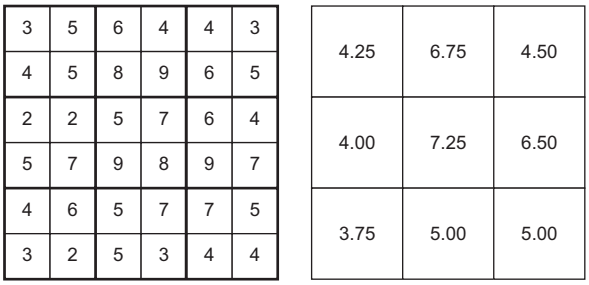


Figure 11.17 Increasing cell size and using the average value of the input pixels as the output value

In the case of figure 11.17, you need four numbers to calculate the output value. To accomplish the same thing with slices, you'd need four slices, with each one corresponding to one of the four input values. Unlike the slices you used for moving windows, however, these would each be much smaller than the original array. Instead, they'd be the same size as the output array, and each one would contain one value per output cell, as shown in figure 11.18. The figure shows the original data, but the cells

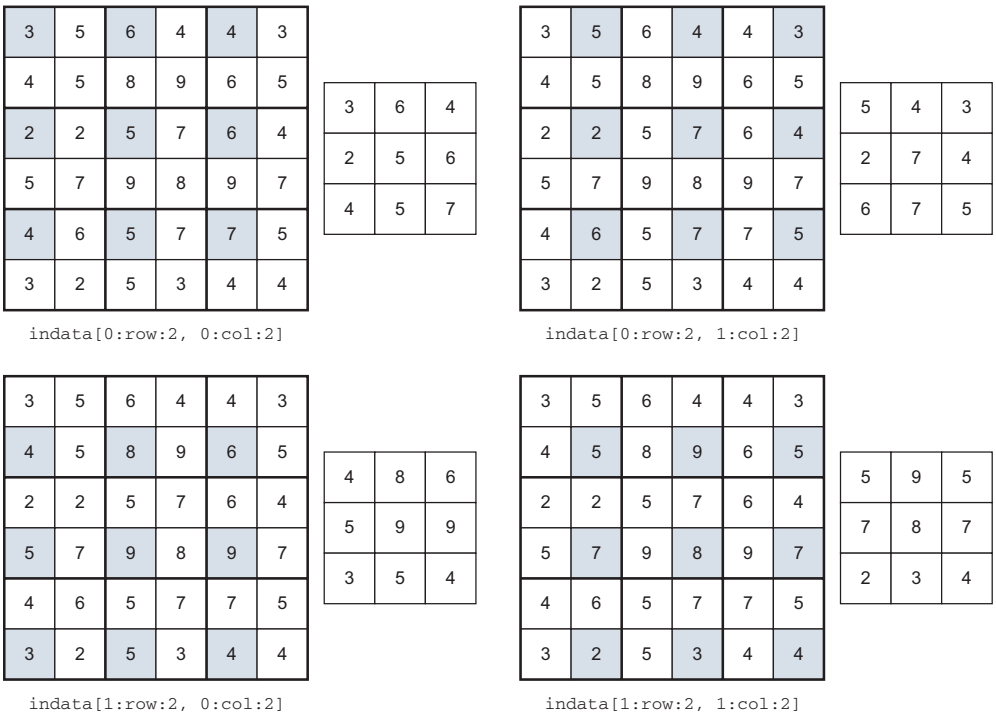


Figure 11.18 Slices used to resample using an average of the input values. The shaded cells are the ones used to create each smaller slice. The smaller arrays are averaged together to get the final result.

that would make up each slice are highlighted. One slice would contain the upper-left pixel from every set of four pixels used to calculate the output. Another slice would contain each upper-right pixel, and so on. Each of the slices in this example has three rows and three columns, which is the same size as the output. If you take the average of these slices on a pixel-by-pixel basis, you end up with the values shown in figure 11.17. For example, the upper-left corner would have a value of (3 + 5 + 4 + 5) / 4 = 17 /4 = 4.25.

Let's look at how to implement this with code. Once again, your life will be easier if you write a function to create the required slices. The following listing shows one that returns the slices in a list, given the original data and the window size (2 x 2 in the example from figure 11.18).

Listing 11.11 Function to make stepped slices

```
def make_resample_slices(data, win_size):
    """Return a list of resampled slices given a window size.

    data     - two-dimensional array to get slices from
    win_size - tuple of (rows, columns) for the input window
    """
    row = int(data.shape[0] / win_size[0]) * win_size[0]
    col = int(data.shape[1] / win_size[1]) * win_size[1]
    slices = []

    for i in range(win_size[0]):
        for j in range(win_size[1]):
            slices.append(data[i:row:win_size[0], j:col:win_size[1]])
    return slices
```

The first thing this function does is calculate the last row and column that will be used. This is necessary because the original data might not be divisible by the size of the window you want. For example, in figure 11.19 the input array has five rows and five columns. The figure shows two of the slices needed to take an average of four pixels, but the second one is smaller than the first. If you try to use these two slices

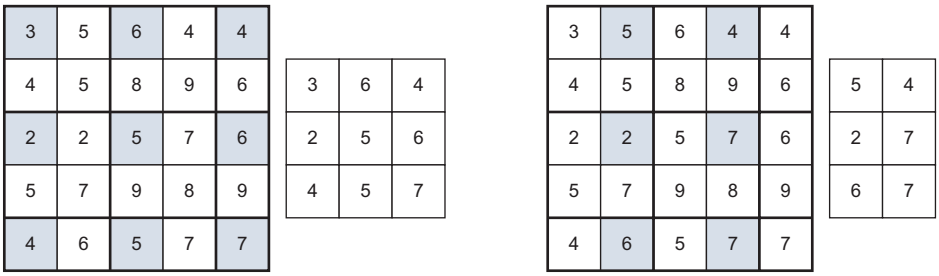


Figure 11.19 When the dimensions of the array to be resampled are not divisible by the dimensions of the window (2 x 2 in this case), the slices are different sizes. This must be accounted for so that the slices have the same size.

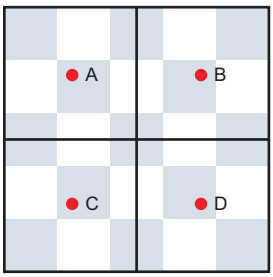
together, you'll get an error because they're different sizes. The function in listing 11.11 handles this by cutting off the fifth row and column, so that the slices are created from four rows and columns instead. To do this, the total number of rows or columns is divided by the number of rows or columns in a window. For the example in figure 11.19, this would be 5 / 2 = 2 for both, so the data can fit two full windows in each direction. Multiply that by the window size to get the total number of rows or columns to use, which is four in the example. This number is used to put an upper bound on the slices.

Once the numbers of rows and columns are known, the function creates one slice for each input location, using the window size as the step parameter so only one input pixel per window is extracted. The slices are all returned as a list, and once you have that you can apply any algorithm you want, as long as you can code it up. To get an average, you could stack the slices and then use the NumPy `mean` function, as you did for moving windows.

TIP Don't forget to change the geotransform to reflect the new cell size when resampling your data.

Techniques like this are great if your output cell size is a multiple of the original, but they don't work in other cases. Let's look at a way to extract specific pixels that can't be specified with a step parameter. To do this, you need to know the original pixel size, the new pixel size, and the numbers of rows and columns in the original image. Get a scaling factor for width by dividing the new pixel width by the original width, and do the same for pixel height. For example, if your original image has a pixel width of 10 but your target width is 25, then the scaling factor is 2.5. The new pixels are 2.5 old pixels wide.

Divide the scaling factor in half to determine that the center of a new pixel is 1.25 old pixels from the edge. This center point is what you want because you use the center of new pixels to determine which nearby old pixels to use when resampling. To get the center x values for the new pixels in terms of the original cell offsets, create an array that starts at the center (1.25) and increments by the scaling factor (2.5). You need to make sure this array goes up to, but not past, the total number of columns in the original array. Do the same for rows, so that you have two arrays containing x and y offsets. Figure 11.20 shows an example of a few pixels. The alternating shaded areas



Indexes for each point
in terms of the small cells
A: [1.25, 1.25]
B: [1.25, 3.75]
C: [3.75, 1.25]
D: [3.75, 3.75]

Figure 11.20 Resampling to a larger pixel size that's not a multiple of the original size. The alternating shaded areas are the original pixels and the thick outlines show the new ones. The dots are the center points for the new pixels.

are the original pixels (of size 10) and the thick outlines denote the pixels with size 25. The dots are the center points of the large pixels, and the text shows the coordinates for these points in terms of the smaller pixels.

Once you have the lists of x and y offsets, you can use the NumPy `meshgrid` function to get two new arrays that contain all possible coordinates obtained from these values. For example, if your row offsets are (3, 5) and column offsets are (2, 4), then the possible combinations are [(3, 2), (3, 4), (5, 2), (5, 4)], and `meshgrid` would return two four-element arrays, one for the row offsets and one for the columns.

The following listing shows a function that computes the scaling factors, makes the offset arrays, and then creates and returns the coordinate arrays.

Listing 11.12 Function to get new pixel offsets in terms of old pixels

```
def get_indices(source_ds, target_width, target_height):
    """Returns x, y lists of all possible resampling offsets.

    source_ds     - dataset to get offsets from
    target_width  - target pixel width
    target_height - target pixel height (negative)
    """
    source_geotransform = source_ds.GetGeoTransform()
    source_width = source_geotransform[1]
    source_height = source_geotransform[5]
    dx = target_width / source_width
    dy = target_height / source_height
    target_x = np.arange(dx / 2, source_ds.RasterXSize, dx)
    target_y = np.arange(dy / 2, source_ds.RasterYSize, dy)
    return np.meshgrid(target_x, target_y)
```

Once you have coordinates, you can take advantage of the fact that offset lists can be used to extract values from NumPy arrays to get the values of the original pixels that fall directly under the center of the new pixels. This is nearest-neighbor resampling, which uses the value of the closest pixel in the original array and doesn't do any other processing. To do this, you'd extract the values at the calculated indices and be done with it, like this:

```
ds = gdal.Open(fn)
data = ds.ReadAsArray()
x, y = get_indices(ds, 25, -25)
new_data = data[y.astype(int), x.astype(int)]
```

The only trick is that you need to convert the indices to integers or NumPy will complain when you attempt to use them to index an array.

Nearest-neighbor is simple, fast, and one of the few appropriate resampling methods for categorical data, but it's not the greatest choice for continuous data. For these types of data, you usually want to use several surrounding pixels to calculate your new value. You could use an average as you did earlier, or one of several other common

resampling methods. Two examples of these are *bilinear interpolation*, which takes a weighted average of the four closest pixels, and *cubic convolution*, which fits a smooth curve through the 16 nearest pixels and uses that to calculate a new value. You're going to write a function that uses the output from your `get_indices` function to perform bilinear interpolation. The hatched areas in figure 11.21 show which four original pixels are used for each center point.

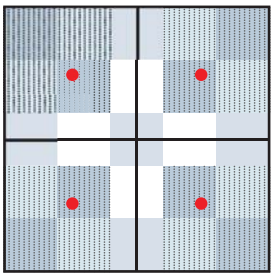


Figure 11.21 The hatched areas show the four original pixels closest to the new pixel center point. These are the ones used to calculate a value for the new pixel.

To get the values of these four pixels, the first thing you need to do is subtract 0.5 from the calculated indices so that they correspond to the center, instead of the edge, of the input pixels. Then you need to determine the integers on either side of these coordinates, which gives you the offsets to use. For example, if the row coordinate is 4.25, then you'd use rows 4 and 5. If you do that for column offsets as well, you have two rows and two columns and can use those to get the four input pixels surrounding the target pixel.

Once you have an input pixel value, multiply it by the distance in both directions from that pixel to the target pixel. This is the part that weights the closer pixels heavier than further pixels. Then add the four weighted values together to get the final output value. If you'd like a more detailed explanation of the algorithm, you can find many sources online.

The following listing shows a function that performs bilinear interpolation, given the original data and the center offsets for the new pixels.

Listing 11.13 Function for bilinear interpolation

```
def bilinear(in_data, x, y):
    """Performs bilinear interpolation.

    in_data - the input dataset to be resampled
    x       - an array of x coordinates for output pixel centers
    y       - an array of y coordinates for output pixel centers
    """
    x -= 0.5
    y -= 0.5
    x0 = np.floor(x).astype(int)
    x1 = x0 + 1
    y0 = np.floor(y).astype(int)
    y1 = y0 + 1

    ul = in_data[y0, x0] * (y1 - y) * (x1 - x)
    ur = in_data[y0, x1] * (y1 - y) * (x - x0)
    ll = in_data[y1, x0] * (y - y0) * (x1 - x)
    lr = in_data[y1, x1] * (y - y0) * (x - x0)

    return ul + ur + ll + lr
```

First and second column offsets

First and second row offsets

Now to use bilinear interpolation to resample a raster, you can use your `get_indices` function to get offsets, which you then pass to the `bilinear` function. Don't forget to edit the geotransform when saving the output, as shown in the following listing.

Listing 11.14 Bilinear interpolation

```
in_fn = r"D:\osgeopy-data\Nepal\everest.tif"
out_fn = r'D:\Temp\everest_bilinear.tif'
cell_size = (0.02, -0.02)

in_ds = gdal.Open(in_fn)
x, y = get_indices(in_ds, *cell_size)
outdata = bilinear(in_ds.ReadAsArray(), x, y)

driver = gdal.GetDriverByName('GTiff')
rows, cols = outdata.shape
out_ds = driver.Create(
    out_fn, cols, rows, 1, gdal.GDT_Int32)
out_ds.SetProjection(in_ds.GetProjection())

gt = list(in_ds.GetGeoTransform())
gt[1] = cell_size[0]
gt[5] = cell_size[1]
out_ds.SetGeoTransform(gt)

out_band = out_ds.GetRasterBand(1)
out_band.WriteArray(outdata)
out_band.FlushCache()
out_band.ComputeStatistics(False)
```

Resample

New image is same size as outdata

Change the geotransform

If you'd like to try other types of interpolation, `scipy.ndimage` contains several interpolation methods. See http://docs.scipy.org/doc/scipy-0.16.1/reference/ndimage.html#module-scipy.ndimage.interpolation for more information.

Resampling with GDAL command-line utilities

This might be a good time to mention the GDAL command-line utilities. There are currently about 30 of them, and new ones get added occasionally. These aren't Python tools; you need to run them from a command prompt or terminal window. Let's see how to use gdalwarp to resample an image. This utility is designed for transforming rasters between spatial reference systems, but you can also use it to resample without changing the spatial reference. The command line looks like this:

```
gdalwarp -tr 0.02 0.02 -r bilinear everest.tif everest_resampled.tif
```

The `-tr` option is for target resolution, in this case indicating that both cell width and height should be 0.02. As you've probably guessed, `-r` stands for resampling method, and this specifies bilinear. Other options include but aren't limited to nearest-neighbor, average, cubic convolution, and mode. The input file is everest.tif, and the new file will be called everest_resampled.tif. Many more options are available, and they're all documented at http://www.gdal.org/gdalwarp.html.

(continued)

If you have GDAL version 2.x, you can also use the same options with the gdal_translate utility, which is designed to convert data between different formats. (I wish I knew how many times I've pointed people to this tool over the years!)

Although these aren't Python tools, you can call them from Python using the subprocess module, which sends commands out to the operating system:

```
import subprocess

args = [
    'gdalwarp',
    '-tr', '0.02', '0.02',
    '-r', 'bilinear',
    'everest.tif', 'everest_resample.tif']
result = subprocess.call(args)
```

The `result` variable will hold 0 if the process completed successfully, and 1 if not. It's preferred that you break your command up into a list of arguments like the example so that Python can handle special cases such as spaces in filenames, but you can also pass a string instead, like this:

```
result = subprocess.call('gdalwarp -tr 0.02 0.02 -r bilinear everest.tif
            everest_resample.tif')
```

11.4 Summary

- If you need to work with large arrays of data in Python, the NumPy module is your answer.
- Use the SciPy module to perform many different scientific data analyses on NumPy arrays.
- Local map algebra computations work on a pixel-by-pixel basis, such as calculating NDVI for a pixel.
- Focal map algebra computations involve a moving window that uses surrounding pixels to calculate the output value, such as calculating slope.
- Zonal calculations work on pixels that are all in the same zone, such as calculating the histogram of landcover types based on land ownership.
- Global calculations, such as proximity analysis, involve the entire raster dataset.