CHAPTER 6

Pycture Perfect Programs

6.1 Objectives

- To understand pixel-based image processing
- To use nested iteration
- To use and understand tuples
- To implement a number of image processing algorithms
- To understand passing functions as parameters

6.2 What Is Digital Image Processing?

Digital photography is a very common way to produce photographs today. It seems that almost everyone has a digital camera as well as software that can organize and manipulate photographs. In this chapter we consider digital images and many of the techniques that can be used to modify them.

Digital image processing refers to the process of using algorithms to edit and manipulate digital images. A **digital image** is a finite collection of small, discrete picture elements called **pixels**. These pixels are organized in a two-dimensional grid and represent the smallest amount of picture information that is available. If you look closely at an image, pixels can sometimes appear as small "dots." More pixels in your image mean more detail or resolution.

Digital cameras are often rated according to how much resolution they provide. Typically resolution is expressed as a number of megapixels. One megapixel means that the picture you take is composed of 1 million pixels. An 8 megapixel camera is capable of taking a picture with up to 8 million pixels.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

6.2.1 The RGB Color Model

Each pixel in the digital image is limited to having a single color. The specific color depends on a formula that mixes various amounts of the primary colors red, green, and blue. Viewing colors as a combination of red, green, and blue is often referred to as the **RGB color model**.

The amount of each primary color component is referred to as its intensity. Intensities will range from a minimum of 0 to a maximum of 255. For example, a color with 255 red intensity, 0 green intensity, and 255 blue intensity will be purple (or magenta). Black will have zero intensity for all primary color components and white will have full color intensity, 255, for all. Table 6.1 shows some common combinations.

An interesting question arises when you consider how many colors there might be using the RGB color model. Since each of the three colors has 256 intensity levels, there are $256^3 = 16{,}777{,}216$ different combinations of red, green, and blue intensities. All of these colors make up the color palette for the RGB color model.

6.2.2 The `cImage` Module

In order to manipulate images we will use a group of objects found in our `cImage` module. (See "Installing the Required Software" (Appendix A) for instructions on downloading and installing `cImage.py`). This module contains objects that allow us to construct and manipulate pixels. We can also construct an image from a file or create a blank image that we can fill in later. In addition, we can create windows where images can be displayed.

Color	Red	Green	Blue
Red	255	0	0
Green	0	255	0
Blue	0	0	255
Magenta	255	0	255
Yellow	255	255	0
Cyan	0	255	255
White	255	255	255
Black	0	0	0

Table 6.1 Red, green, and blue intensities for some common colors

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

Method Name	Example Use	Explanation
`Pixel(r,g,b)`	`p = Pixel(25,200,143)`	Create a pixel with 25 red, 200 green, and 143 blue.
`getRed()`	`r = p.getRed()`	Return the red component intensity.
`getGreen()`	`g = p.getGreen()`	Return the green component intensity.
`getBlue()`	`g = p.getBlue()`	Return the blue component intensity.
`setRed()`	`p.setRed(100)`	Set the red component intensity to 100.
`setGreen()`	`p.setGreen(45)`	Set the green component intensity to 45.
`setBlue()`	`p.setBlue(87)`	Set the blue component intensity to 87.

Table 6.2 `Pixel` object

The `Pixel` Object

Images are collections of pixels. In order to represent a pixel, we need a way to collect together the red, green, and blue components. The `Pixel` object provides a constructor and methods that allow us to create and manipulate the color components of pixels. Table 6.2 shows the constructor and methods provided by pixel objects. Session 6.1 shows them in action. The constructor will require the three color components. It will return a reference to a `Pixel` object that can be accessed or modified. We can extract the color intensities using the `getRed`, `getGreen`, and `getBlue` methods. Similarly, we can modify the individual components within a pixel using `setRed`, `setGreen`, and `setBlue`.

```
>>> from cImage import *
>>> p = Pixel(200,100,150)
>>> p
(200, 100, 150)
>>> p.getRed()
200
>>> p.setBlue(20)
>>> p
(200, 100, 20)
>>>
```

Session 6.1 Creating and using a pixel

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

The ImageWin Object

Before creating images, we will create a window that can be used to display our images. The `ImageWin` object provides a constructor that produces a window with a title, width, and height. When a window is constructed, it is immediately shown. The code below produces an empty window that is 600 pixels wide and 400 pixels high. Table 6.3 shows additional methods for the `ImageWin` object. Note that the `getMouse` method returns a coordinate position within the window itself and is not related to any particular image that might be displayed within the window.

```
>>> from cImage import *
>>> myWin = ImageWin("Image Processing",600,400)
>>>
```

Method Name	Example Use	Explanation
`ImageWin(title, width, height)`	`ImageWin("Pictures", 800, 600)`	Create a window to display images that are 800 pixels wide and 600 pixels high.
`exitOnClick()`	`myWin.exitOnClick()`	Close the image window and exit when the mouse is clicked.
`getMouse()`	pos = `myWin.getMouse()`	Return an (x, y) tuple representing the position of the mouse click in the window.

Table 6.3 The `ImageWin` object

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

Method Name	Example Use	Explanation
`FileImage(filename)`	im = `FileImage( "pic.gif")`	Create an image object from a file named pic.gif.
`EmptyImage(width, height)`	`im = EmptyImage(300, 200)`	Create an empty image that is 300 pixels wide and 200 pixels high.
`getWidth()`	`w = im.getWidth()`	Return the width of the image in pixels.
`getHeight()`	`h = im.getHeight()`	Return the height of the image in pixels.
`getPixel(col, row)`	`p = im.getPixel(150, 100)`	Return the `Pixel` from row 100, column 150.
`setPixel(col, row, newp)`	`im.setPixel(150, 100, Pixel(255, 255, 255))`	Set the pixel at row 100, column 150 to be white.
`setPosition(col, row)`	`im.setPosition(20, 20)`	Position the top-left corner of the image at (col, row) in the window.
`draw(imagewin)`	`im.draw(myWin)`	Draw the image `im` in the `myWin` image window. It will default to the upper-left corner.
`save(fileName)`	`im.save(fileName)`	Save the image to a file. Use `gif` or `ppm` as the extension.

Table 6.4 `FileImage` and `EmptyImage` Objects

Now that we have a window, we need to create images to display. The `cImage` module provides two kinds of image objects: `FileImage` and `EmptyImage`. These objects allow us to create and manipulate images, and they give us simple access to the pixels in each image. Table 6.4 shows the two constructors for creating images as well as other methods that are provided by both objects.

The `FileImage` Object

The `FileImage` object is an image that is constructed from files such as those that are created by digital cameras or that reside on web pages. For example, the file `lutherBell.gif`

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

was taken with an ordinary digital camera. Session 6.2 shows that the `FileImage` constructor needs only the name of the image file. It converts the image stored in that file to an image object. In this case, `bell` is the reference to that object. Finally, we can use the `draw` method to ask the image object to show itself in an image window. The default positioning is to place the image in the upper-left corner of the window. Figure 6.1 shows the result.

```
>>> from cImage import *
>>> myWin = ImageWin("Luther Bell",300,200)
>>> bell = FileImage("lutherBell.gif")
>>> bell.draw(myWin)
>>>
```

Session 6.2 Creating and showing a file image

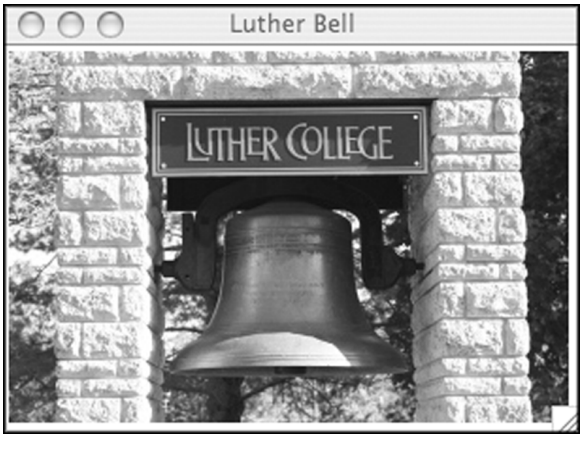

Figure 6.1 Drawing an image in a window

As we said earlier, an image is a two-dimensional grid of pixel values. Each small square in Figure 6.2 represents a pixel that can take on any one of the millions of colors in the RGB color model. We can access information about our specific image by using the `getWidth` and `getHeight` methods (see Session 6.3). This image is 300 pixels across and 200 pixels top to bottom. Rows are numbered from 0 to 1 less than the height of the image. Columns are numbered from 0 to 1 less than the width.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

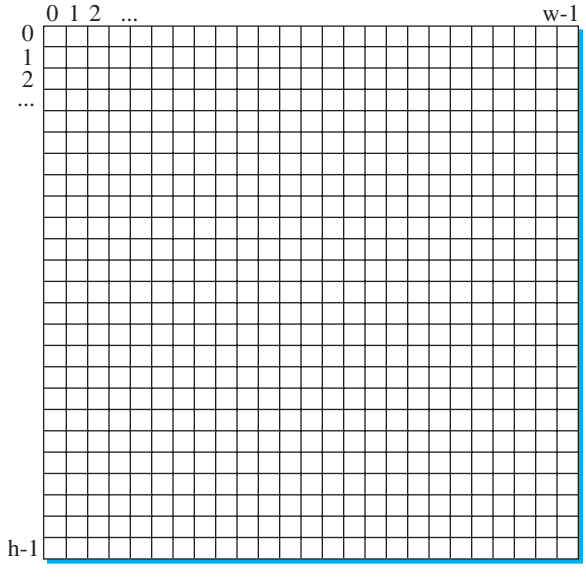

Figure 6.2 Details of an image

```
>>> bell.getWidth()
300
>>> bell.getHeight()
200
>>> bell.getPixel(124,165)
(117, 123, 123)
>>>
```

Session 6.3 Accessing information about an image

We can access a particular pixel by using the `getPixel` method. To use `getPixel`, we need to provide the location for the pixel of interest. The location will be a pair of values, one specifying the column and one specifying the row. Each unique column-row pair will provide access to a single pixel.

In the example, we are accessing the pixel located at column 124, row 165. In other words, it is the pixel that is 125 pixels from the left and 166 pixels from the top. It is important to remember that we start counting at 0. The value that is returned in our example shows the red, green, and blue components of the `Pixel` at that location in our image.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

The `EmptyImage` Object

We often want to build a new image pixel by pixel starting with a "blank" image. Using the `EmptyImage` constructor, we can create an image that has a specific width and height but where each pixel is void of color. This means that each `Pixel` has the value `(0,0,0)` or "black." The statements in Session 6.4 create an empty image with all black pixels.

```
>>> myImWin = ImageWin("Empty Image", 300, 300)
>>> emptyIm = EmptyImage(300,300)
>>> emptyIm.draw(myImWin)
>>>
```

Session 6.4 Creating and displaying an empty image

As an example to show the basic use of the image methods, we will construct an image that starts out empty and is filled with white pixels at specific locations. Session 6.5 starts by creating a window and an empty image that is sized to fit within the window. In order to create a line of white pixels, we use a loop variable `i` and iterate over the range from 0 to 299. The `setPixel` method can be called using the value of `i` for both column and row with a pixel called `whitePixel` that has been created with a combination of red, green, and blue corresponding to the color `white`. We draw the image in the window as shown in Figure 6.3. Finally, we save the image to a file using the `save` method.

```
>>> from cImage import *
>>> myImWin = ImageWin("Line Image",300,300)
>>> lineImage = EmptyImage(300,300)
>>> whitePixel = Pixel(255,255,255)
>>> for i in range(300):
        lineImage.setPixel(i,i,whitePixel)

>>> lineImage.draw(myImWin)
>>> lineImage.save("lineImage.gif")
>>>
```

Session 6.5 Using `EmptyImage`

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

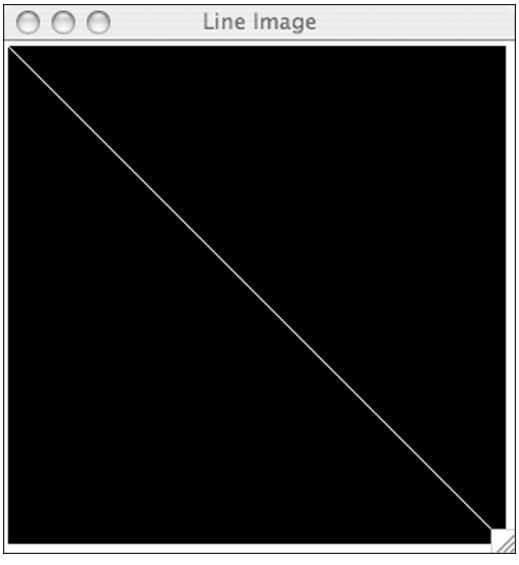

Figure 6.3 Creating a white diagonal line

Exercises

6.1 Modify Session 6.5 to create a line with random pixel colors.

6.2 Write a function to create an image of a rectangle. Start with an `EmptyImage`. *Challenge*: Create a filled rectangle.

6.3 Write a function to create an image of a circle.

6.4 Download an image from the Web or your personal image collection and display it in a window. Note the image must be in gif format.

6.5 Modify some pixels in the image from the previous question and save it.

6.6 Why is the number of color intensities limited to the range 0–255? Do some research to find the answer to this question.

6.3 Basic Image Processing

We now have all of the tools necessary to do some simple image processing. Our first examples will perform color manipulations on an image. In other words, we want to take the existing pixels and modify them in some way to change the appearance of the original

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

image. The basic idea will be to systematically process each pixel one at a time and perform the following operations:

1. Extract the color components.
2. Build a new pixel.
3. Place that pixel in a new image at the same location as the original.

Once we see the general pattern, the options are endless. Note that all of the newly constructed images in this section will be the same dimensions as the image they are based on.

6.3.1 Negative Images

When images are placed on film and then developed, a set of **negatives** is produced. A negative image is also known as a color-reversed image. In a negative image, red becomes cyan where cyan is the mixture of green and blue. Likewise, yellow becomes blue and blue becomes yellow. Regions that are white turn black, black turns white, light turns dark and dark turns light. This continues for all possible color combinations.

At the pixel level, the negative operation is just a matter of "reversing" the red, green, and blue components in that pixel. Since color intensity ranges from 0 to 255, a pixel with a large amount of a specific color—say, red—will have a small amount in the negative. At the maximum, a pixel with a red intensity of 255 will have a red intensity of 0 in the negative. This suggests that the way to create a negative pixel is to subtract each of the red, green, and blue intensity values from 255. The results can then be placed in a new pixel.

Listing 6.1 shows a function that will take a `Pixel` as a parameter and return the negative pixel using the suggested process from above. Note that the function expects to receive an entire `Pixel` and will decompose the color components, perform the subtractions, and then build and return a new `Pixel`. We can easily test this function as shown in Session 6.6.

```
1 def negativePixel(oldPixel):
2     newred = 255 - oldPixel.getRed()
3     newgreen = 255 - oldPixel.getGreen()
4     newblue = 255 - oldPixel.getBlue()
5     newPixel = Pixel(newred, newgreen, newblue)
6     return newPixel
```

Listing 6.1 Constructing a negative pixel

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

```
>>> apixel = Pixel(155,23,230)
>>> negativePixel(apixel)
(100, 232, 25)
>>>
```

Session 6.6 Testing the `negativePixel` function

In order to create the negative image, we call the `negativePixel` function on each pixel. We need to come up with a pattern that will allow us to process each pixel. To do this, we can think of the image as having a specific number of rows equal to the height of the image. Each row in turn has a number of columns equal to the width of the image. With this in mind, we can build an iteration that will systematically move through all of the rows and within each, will move through all of the columns. This gives rise to the notion of **nested iteration**—the placement of an iteration as the process inside of another iteration. In other words, for each pass of the "outer" iteration, the "inner" iteration will run to completion. The inner iteration will run from start to finish for each pass of the outer iteration.

As an example, consider the code fragment in Session 6.7 using *for* statements. The outer iteration is moving over the list `[0,1,2]`, which was produced by `range(3)`. For each item in that list, the inner loop will iterate over the characters `'c'`, `'a'`, `'t'`. This means that the `print` function will be called for each character in the string `"cat"` for each number in the list `[0,1,2]`.

```
>>> for num in range(3):
        for ch in "cat":
            print(num,ch)


0 c
0 a
0 t
1 c
1 a
1 t
2 c
2 a
2 t
>>>
```

Session 6.7 Showing nested iteration with lists and strings

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

The resulting output shows nine lines. Each group of three represents one pass of the outer loop. Within each group, the value of `num` stays the same. For each value of `num`, the entire inner loop completes, and therefore each of the three characters of the string appears.

We can now apply this idea to the construction of a function to compute the negative of each pixel in an image (see Listing 6.2). The function will take a single parameter that gives the name of a file containing an image. The function will not return anything but will simply display both the original and the negative image.

The first step (lines 2–8), is to create an image window, open an original image, and draw it in the window. We then need to create an empty image that has the same width and height as the original. Note that `width` and `height` are the width and height of both the original and new image.

Using the idea of nested iteration, we will first iterate over the rows, starting with row 0 and extending down to `height-1`. For each row, we will process all of the columns within the row.

```
for row in range(height):
   for col in range(width):
```

Each pixel is accessed by `row` and `col`. We can get the original color tuple at that location using the `getPixel` method (line 12). Once we have the original pixel, we can use the `negativePixel` function to transform it into the negative. Finally, using the same `row` and `col`, we can place the new negative pixel in the new image (line 14) using the `setPixel` method. Once the iteration is complete for all pixels, the new image is drawn in the window. Note that we use the `setPosition` method to place the new image next to the original. Figure 6.4 shows the resulting images.

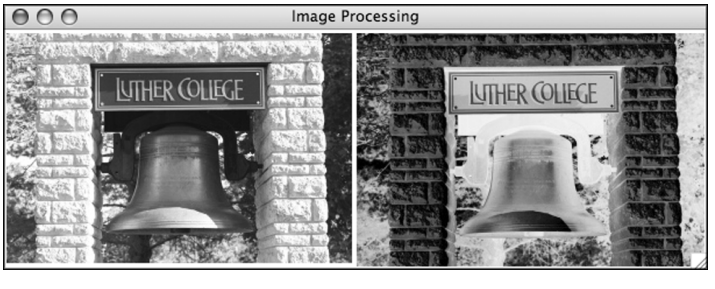

Figure 6.4 A negative image

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

```
def makeNegative(imageFile):
    myimagewindow = ImageWin("Image Processing",600,200)
    oldimage = FileImage(imageFile)
    oldimage.draw(myimagewindow)

    width = oldimage.getWidth()
    height = oldimage.getHeight()
    newim = EmptyImage(width,height)

    for row in range(height):
        for col in range(width):
            originalPixel = oldimage.getPixel(col,row)
            newPixel = negativePixel(originalPixel)
            newim.setPixel(col,row,newPixel)

    newim.setPosition(width+1,0)
    newim.draw(myimagewindow)
    myimagewindow.exitOnClick()
```

Listing 6.2 Constructing a negative image

6.3.2 Grayscale

Another very common color manipulation is to convert an image to **grayscale**, where each pixel will be a shade of gray, ranging from very dark (black) to very light (white). With grayscale, each of the red, green, and blue color components will take on the same value. In other words there are 256 different grayscale color values ranging from darkest (0, 0, 0) to lightest (255, 255, 255). The standard color known as "gray" is typically coded as (128, 128, 128).

Our task then is to take each color pixel and convert it into a gray pixel. The easiest way to do this is to consider that the intensity of each red, green, and blue component needs to play a part in the intensity of the gray. If all of the color intensities are close to zero, the resulting color is very dark, which should in turn show as a dark shade of gray. On the other hand, if all of the color intensities are closer to 255, the resulting color is very light and the resulting gray should be light as well.

This analysis gives rise to a simple but accurate formula for grayscale. We will simply take the average intensity of the red, green, and blue components. This average can then be used for all three color components in a new pixel that will be a shade of gray. Listing 6.3 shows a function, similar to the `negativePixel` function described previously, that takes a `Pixel` and returns the grayscale equivalent. Session 6.8 shows the function in use.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

```
1 def grayPixel(oldpixel):
2     intensitySum = oldpixel.getRed() + oldpixel.getGreen() + \
3                    oldpixel.getBlue()
4     aveRGB = intensitySum // 3
5
6     newPixel = Pixel(aveRGB,aveRGB,aveRGB)
7     return newPixel
```

Listing 6.3 Constructing a grayscale pixel

```
>>> grayPixel( Pixel(34,128,74) )
(78, 78, 78)
>>> grayPixel( Pixel(200,234,165) )
(199, 199, 199)
>>> grayPixel( Pixel(23,56,77) )
(52, 52, 52)
>>>
```

Session 6.8 Testing the `grayPixel` function

Now the process of creating a grayscale image can proceed in the same way as described for creating the negative (see Listing 6.4). After opening and drawing the original image, we create a new, empty image. Using nested iteration, process each pixel, this time converting each to the corresponding grayscale value (line 13). The final image is shown in Figure 6.5.

We developed the previous examples by continually building upon a framework of simple ideas. We started with the pixel, then created a function to transform the color components

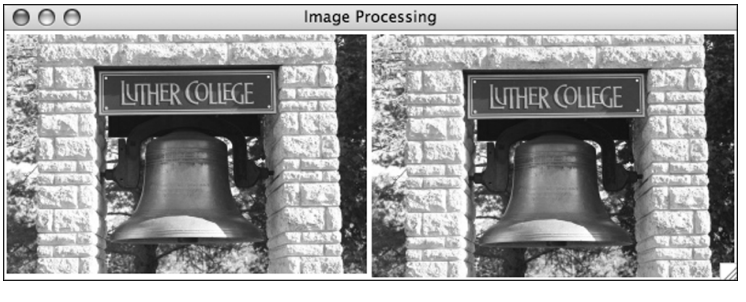

Figure 6.5 A grayscale image

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

```
def makeGrayScale(imageFile):
    myimagewindow = ImageWin("Image Processing",600,200)
    oldimage = FileImage(imageFile)
    oldimage.draw(myimagewindow)

    width = oldimage.getWidth()
    height = oldimage.getHeight()
    newim = EmptyImage(width,height)

    for row in range(height):
        for col in range(width):
            originalPixel = oldimage.getPixel(col,row)
            newPixel = grayPixel(originalPixel)
            newim.setPixel(col,row,newPixel)

    newim.setPosition(width+1,0)
    newim.draw(myimagewindow)
    myimagewindow.exitOnClick()
```

Listing 6.4 Constructing a grayscale image

of a pixel, and finally applied that function to all of the pixels in the image. This stepwise construction is a very common methodology used in writing computer programs. Building upon those functions that already work in order to provide more complex functionality that can again be used as a foundation allows programmers to be very efficient. We take this one step further in the next section.

Exercises

6.7 Write a function that removes all red from an image.

6.8 Write a function that enhances the red intensity of each pixel in an image.

6.9 Write a function that diminishes the blue intensity of each pixel in an image.

6.10 Write a function that manipulates all three color intensities in a pixel using a strategy of your own choice.

6.11 Write a function that takes a color image and displays a black and white image next to it. *Hint:* You may want to start by converting the image to grayscale. Any pixel with a gray value less than some threshold will become black. All other pixels will be white.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

6.3.3 A General Solution: The Pixel Mapper

If you compare the Python listings for the `makeGrayScale` and `makeNegative` functions, you will note that there is quite a bit of redundancy. In fact, the same steps were followed with only one exception—namely, the function that was called to map each original pixel into a new pixel. This similarity causes us to think that we could factor out the code that is the same and create a more general Python function. This is another example of using abstraction to solve problems.

Figure 6.6 shows how such a function might be constructed. We will create a function called `pixelMapper` that will take two parameters, an original image and an RGB function. The `pixelMapper` function transforms the original image into a new image using the RGB function. After applying the RGB function to every pixel, the transformed image is returned. In this way we have a single function that is capable of transforming an image given any function that manipulates the color intensities of a single pixel.

In order to implement this general pixel mapper, we need to be able to pass a function as a parameter. Up to this point, all of our parameters have been data objects such as integers, floating point numbers, lists, tuples, and images. The question to consider is whether there is any difference between a function and a typical data object.

The answer to this question is that there is no difference. To understand why, we will first look at a simple example. The function `squareIt` takes a number and returns the square.

```
def squareIt(n):
   return n * n
```

We can invoke the `squareIt` function with the usual syntax (see Session 6.9), placing the actual value to be squared as a parameter. However, if we evaluate the name of the function without invoking it (without the parameters in parentheses), we can see that the result is a function definition. The name of a Python function is a reference to a data object—in particular, a function definition (see Figure 6.7). Note that the strange looking "number," `0x1021730`, is actually the address where the function is stored in memory.

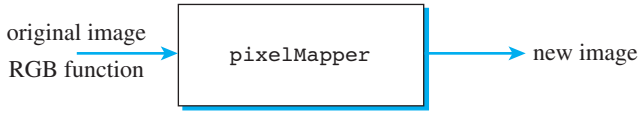

Figure 6.6 A general pixel mapping function

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

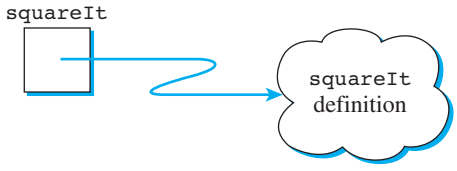

Figure 6.7 A function is a data object

Since a function is simply another data type, you might wonder what kinds of operators you can use with it. In fact there are only two operators that you can use with a function. The parentheses are actually an operator that tells Python to apply the function to the supplied parameters. In addition, since a function is an object, you can use the assignment operator to give a function another name, as shown in Session 6.9. Note that now the variable `z` is a reference to the same data object as `squareIt` and can be used with the parenthesis operator.

```
>>> squareIt(3)
9
>>> squareIt(squareIt(3))
81
>>> squareIt
<function squareIt at 0x1021730>
>>> z = squareIt
>>> z(3)
9
>>> z
<function squareIt at 0x1021730>
```

Session 6.9 Evaluating the `squareIt` function

Since any Python object can be passed as a parameter, it is certainly possible to pass the function definition object. We just need to be careful not to invoke the function prior to passing it. To show this (Session 6.10), we create a simple function called `test` that expects two parameters, a function object and a number. The body of `test` will invoke the function object using the number as a parameter and return the result.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

```
>>> def test(funParm, n):
        return funParm(n)
>>>
>>> test(squareIt,3)
9
>>> test(squareIt,5)
25
>>> test(squareIt(3),5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in test
TypeError: 'int' object is not callable
```

Session 6.10 Using a function passed as a parameter

We can then use our `test` function by passing the `squareIt` function definition. In addition, we will pass the integer 3. Remember, when we pass the function definition object, we do not include the parentheses pair. Figure 6.8 shows the references immediately after `test` has been called and the parameters have been received. A copy of the reference to the actual parameter `squareIt` is received by `funParm` and `n` contains a reference to the object 3.

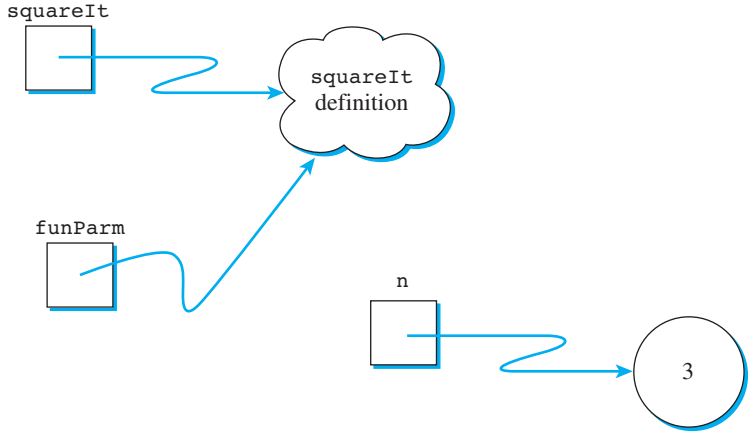

Figure 6.8 Passing the method and an integer

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

The statement `return funParm(n)` will cause the function referred to by `funParm` to be invoked on the value of `n`. In this case, since `funParm` is a reference to `squareIt`, the `squareIt` function will be invoked with the value of 3. The next example shows the result of passing 5 instead of 3. Note the error message at the end of the session when we call the `squareIt` function instead of passing the definition.

With these mechanics in place, it is now possible to create an implementation for our general `pixelMapper` function (see Listing 6.5). Recall that this function will take two parameters, an image and an RGB function. Lines 3–5 construct a new empty image that is the same size as the original. Line 10, which is inside of our nested iteration, does most of the work. The `rgbFunction` parameter is applied to each pixel from the original image and the resulting pixel is placed in the new image. Once the nested iteration is complete, we will return the new image.

```
1  def pixelMapper(oldimage,rgbFunction):
2
3      width = oldimage.getWidth()
4      height = oldimage.getHeight()
5      newim = EmptyImage(width,height)
6
7      for row in range(height):
8          for col in range(width):
9              originalPixel = oldimage.getPixel(col,row)
10             newPixel = rgbFunction(originalPixel)
11             newim.setPixel(col,row,newPixel)
12
13     return newim
```

Listing 6.5 A general pixel mapping method

We can complete our example by calling the `pixelMapper` function using one of our RGB functions from the previous sections. Listing 6.6 shows a main function, `generalTransform`, that sets up the image window and loads the original image. Line 6 invokes `pixelMapper` using the `grayPixel` function. The result is identical to that shown in Figure 6.5.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

```
def generalTransform(imageFile):
    myimagewindow = ImageWin("Image Processing",600,200)
    oldimage = FileImage(imageFile)
    oldimage.draw(myimagewindow)

    newimage = pixelMapper(oldimage,grayPixel)
    newimage.setPosition(oldimage.getWidth()+1,0)
    newimage.draw(myimagewindow)
    myimagewindow.exitOnClick()
```

Listing 6.6 Calling the general pixel mapping function

Exercises

6.12 Use a reference diagram to explain the error message in Session 6.10.

6.13 Use the `generalTransform` and `pixelMapper` functions to create a negative image using the `negativePixel` function developed earlier.

6.14 Write an RGB function to remove the red from a pixel. Test this function with `pixelMapper`.

6.15 Write an RGB function to convert a pixel to black and white. Test this function with `pixelMapper`.

6.16 Write an RGB function of your choice. Test this function with `pixelMapper`.

6.17 Sepia tone is a brownish color that was used for photographs in times past. The formula for creating a sepia tone is as follows:

$$\begin{aligned} \texttt{newR} &= (\texttt{R} \times 0.393 + \texttt{G} \times 0.769 + \texttt{B} \times 0.189) \\ \texttt{newG} &= (\texttt{R} \times 0.349 + \texttt{G} \times 0.686 + \texttt{B} \times 0.168) \\ \texttt{newB} &= (\texttt{R} \times 0.272 + \texttt{G} \times 0.534 + \texttt{B} \times 0.131) \end{aligned}$$

Write an RGB function to convert a pixel to sepia tone. *Hint:* Remember that RGB values must be integers between 0 and 255.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

6.4 Parameters, Parameter Passing, and Scope

Throughout many of the previous chapters, we have used functions to implement abstraction. We have broken problems down into smaller, more manageable pieces and have implemented functions that we can call over and over again. In the previous section we pushed this idea one step further by passing functions as parameters to other functions. In this section we explore in more detail the underlying mechanics of how functions and parameter passing work.

Consider the function shown in Listing 6.7, which computes the hypotenuse of a right triangle. Using the **Pythagorean theorem**, $a^2 + b^2 = c^2$, this function needs the lengths of the two sides, called `a` and `b`, and computes and returns the length of the long side, called `c`. Session 6.11 shows the function in use.

```
1 import math
2 def hypotenuse(a,b):
3     c = math.sqrt(a**2 + b**2)
4     return c
```

Listing 6.7 A simple function to compute the hypotenuse of a triangle

```
>>> hypotenuse(3,4)
5.0
>>>
>>> side1 = 3
>>> side2 = 4
>>> hypotenuse(side1,side2)
5.0
>>>
>>> hypotenuse(side1*2, side2*2)
10.0
>>>
>>> hypotenuse
<function hypotenuse at 0x42b70>
```

Session 6.11 A simple method

In the first example, references to the objects 3 and 4 are passed to the function. These are known as the **actual parameters** as they represent the "actual" data that the function will receive. As we have seen before, the parameter list `(a,b)` receives these object references

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

one at a time, in order, from left to right. So, `a` receives a reference to the object `3` and `b` receives a reference to the object `4`. These parameters, defined in the function itself, are known as the **formal parameters**.

In the second example shown, the actual parameters are not literal numbers but are instead names that are referring to the objects 3 and 4. Prior to the function call, Python evaluates the two names `side1` and `side2` in order to find the objects. Once again, `a` receives a reference to the object `3` and `b` receives a reference to the object `4`.

6.4.1 Call by Assignment Parameter Passing

In general, the process by which a function's formal parameter receives an actual parameter value is known as **parameter passing**. There are many different ways to pass parameters and different programming languages have chosen to use a variety of them. In Python, however, all parameters are passed using a single mechanism known as **call by assignment** parameter passing.

Call by assignment parameter passing uses a simple two-step process to pass data when the function is called. Calling a function is also known as **invocation**. The first thing that happens is that the actual parameters are evaluated. This evaluation results in an object reference to the result. In the first case from Session 6.11, evaluating a literal number simply returns a reference to the number itself. In the second example, evaluating a variable name returns the object reference named by that variable.

Once the evaluation of the actual parameters is complete, the object references are passed to and received by the formal parameters in the function. The formal parameter becomes a new name for the reference that is passed. In a sense it is as if we executed the assignment statement `formal parameter = actual parameter`.

As a final example, consider the third invocation shown in Session 6.11. Here the actual parameters are expressions that double the lengths of the original sides. Call by assignment parameter passing evaluates these expressions first and then assigns the references to the resulting objects to the formal parameters `a` and `b`. The `hypotenuse` function has no idea where the references came from or how complicated the original expression might have been. The references that are received are simply the results of the evaluation.

Call by assignment parameter passing has some important ramifications that may not be obvious to you at first. Changes to the formal parameter may or may not result in changes to the actual parameter depending on whether the actual parameter is mutable or immutable. If the actual parameter is immutable, then changes to the formal parameter will have no effect outside the function. If the actual parameter is mutable, then changes to the object referenced by the formal parameter will be visible outside the function.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

For example, if the actual parameter is a reference to the integer 3, then assigning a reference to the integer 5 to the formal parameter would not be visible outside the function. If the actual parameter is a reference to a list, then any changes to the contents of the list including additions or deletions will be visible outside the function. However, if the formal parameter is assigned to a different list, the behavior is consistent with the behavior for integers.

6.4.2 Namespaces

In Python, all of the names defined in a program, whether for data or for functions, are organized into **namespaces**—collections of all names that are available to be accessed at a particular point in time during the execution of a Python program. When we start Python, we create two namespaces. The first is called the **built-in namespace** (see Figure 6.9), and it includes all of the system-defined names of functions and data types that we regularly use in Python. Names such as `range`, `str`, and `float` are included in this namespace. The second is called the **main namespace**, and it is initially empty. Python calls these two namespaces `__builtin__` and `__main__`.

As we begin to create our own names in a Python session, they are added to the main namespace. As shown in the example in Session 6.11, the variable names `side1` and `side2` are now added to the main namespace. The function name `hypotenuse` is also added to the main namespace. Note that the names are added as a result of an assignment statement or a method definition. The objects referred to by the names are shown as well. Figure 6.10 shows the location of the names and the objects they reference. Note that the objects exist outside the namespace.

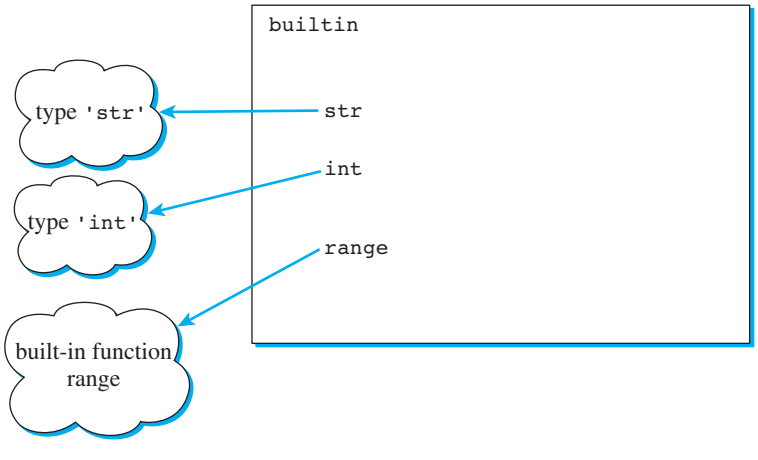

Figure 6.9 The `builtin` namespace

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

At this point it is instructive to investigate the namespace idea interactively. Python provides us with the function `dir` that allows us to see a list of the names in any namespace. Session 6.12 shows what happens when we invoke the `dir` function after defining the `hypotenuse` function and assigning values to the `side1` and `side2` variables.

In addition to the three names that we just created, there are four other names of interest defined in our namespace. The `__builtins__` refers to the built-in namespace we referred to earlier. It is possible for you to find out the names of the objects in the built-in namespace by calling the function `dir(__builtins__)`. The `__doc__` name is always available to hold a reference to a string that describes the namespace. The `__main__` namespace does not have documentation but other namespaces such as math may. `__name__` holds the name of the current namespace. In Session 6.12 you can see that we evaluated `__name__` to find that we are in the `__main__` namespace.

The final name in the list from `dir` that we have yet to discuss is `math`. This name appears because we imported the math module on line 1 of the session. As you can see, the name `math` refers to an object that is a module. It is important to note that a module defines its own namespace. Just as with the built-in namespace, you can find out the names in the math namespace by evaluating `dir(math)`. Note that the math namespace also has its own `__name__` and `__doc__` entries.

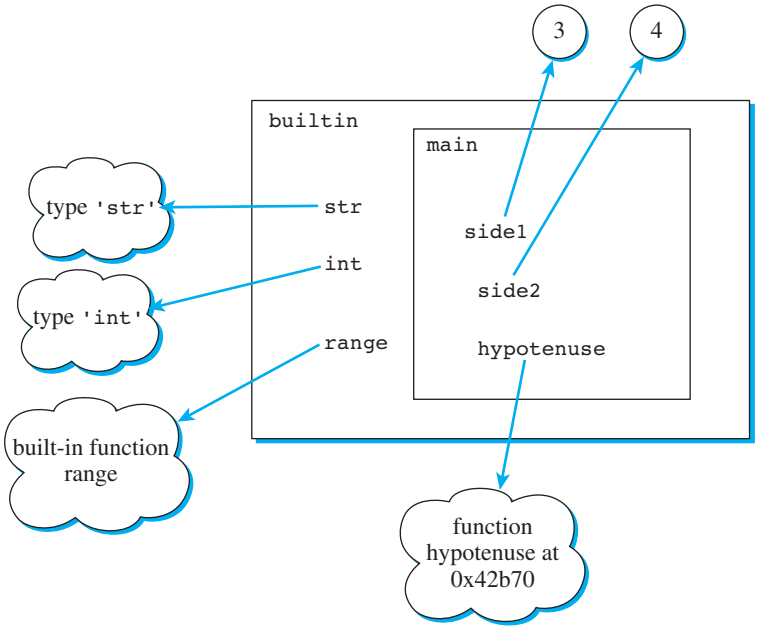

Figure 6.10 The `main` namespace

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

```
>>> import math

>>> def hypotenuse(a,b):

...     c= math.sqrt(a**2+b**2)

...     return c

...

>>> side1 = 3

>>> side2 = 4

>>> dir()

['__builtins__', '__doc__', '__name__', '__package__', 'hypotenuse', 'math',
'side1', 'side2']

>>> __name__

'__main__'

>>> math

<module 'math' from '.../lib/python3.0/lib-dynload/math.so'>

>>> dir(math)

['__doc__', '__file__', '__name__', '__package__', 'acos', 'asin', 'atan',
'atan2', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'exp', 'fabs',
'floor', 'fmod', 'frexp', 'hypot', 'isinf', 'isnan', 'ldexp', 'log', 'log10',
'modf', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']

>>> math.__doc__

'This module is always available.  It provides access to the\nmathematical
functions defined by the C standard.'

>>>
```

Session 6.12 Exploring the `__main__` namespace

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

We now have all the tools needed to understand the difference between the statements `import math` and `from math import *`. Session 6.13 shows the result of calling the `dir` function after importing the math module using the statement `from math import *`. Notice that all the names from the math module now appear as part of the main namespace. This allows us to call functions such as `sqrt` directly without prefacing the function name with the module name.

```
>>> from math import *

>>> dir()

['__builtins__', '__doc__', '__name__', '__package__', 'acos', 'asin', 'atan',
'atan2', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'exp', 'fabs',
'floor', 'fmod', 'frexp', 'hypot', 'isinf', 'isnan', 'ldexp', 'log', 'log10',
'modf', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']

>>>
```

Session 6.13 Importing the math names into the `__main__` namespace

Exercises

6.18 Try calling the `dir` function on the `__builtins__` object.

6.19 Import the `turtle` module and find out the names defined.

6.20 Look at the `__doc__` string for the `turtle` module.

6.21 If you put a string at the beginning of any Python file, that string becomes the `__doc__` string for that module. Try adding a string to the beginning of one of your Python files. Can you import that file and see the string you added? The `help` function also returns this string as part of the documentation for a module.

6.4.3 Calling Functions and Finding Names

When a function is invoked, a new namespace known as the **local namespace** is created corresponding to the function itself. This namespace includes those names that are created inside the function. This includes the formal parameters as well as any names that are

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

used on the left-hand side of an assignment statement in the body of the function. These names are referred to as **local variables** since they have been created within the function and are part of the local namespace. When the function is completed, either due to a `return` statement or simply due to running out of code statements, the local namespace is destroyed. This means that all of the locally defined names are no longer available for use.

It is important to note the placement of these namespaces with respect to one another. The main namespace is placed inside the built-in namespace. Likewise, local namespaces are placed inside the namespace of the module where they are defined. For programs that you write, the namespaces for your functions will be placed in the main namespace. The namespaces for functions from modules that you import will be placed in the namespace of the imported module. Figure 6.11 shows the placement of the local namespace for the `hypotenuse` function when it is called. This figure also provides an illustration of the call by assignment mechanism. Note that the formal parameters `a` and `b` are referring to the same objects as the actual parameters `side1` and `side2`.

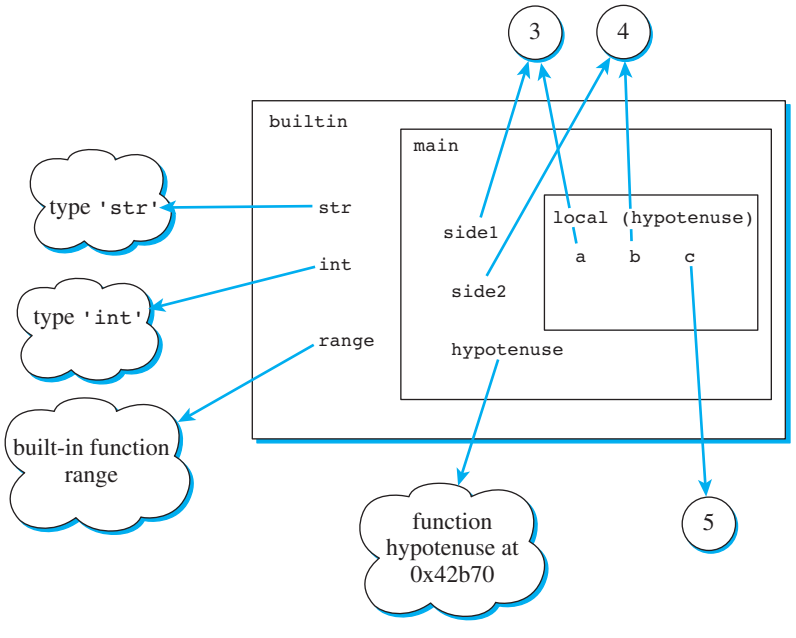

Figure 6.11 A local namespace

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

When a name is used in a statement, Python needs a way to locate the particular occurrence of that name among all of the names that have been introduced up to that point. In order to find the name, Python uses the following simple rules:

1. Whenever a name is used, except on the left-hand side of an assignment statement, Python searches through the namespaces in the following order:

   (a) The current local namespace, if one exists

   (b) The main or module namespace

   (c) The built-in namespace

   When Python finds the first occurrence of the name, the search ends. Looking again at Figure 6.11, you may find it helpful to think of this as searching from the "inside out." If the name is not found, an error is reported.

2. When you use a name on the left-hand side of an assignment statement, Python searches only the current namespace.

   (a) If the name is not found, a new name is created in the current namespace.

   (b) If the name is found, then the old reference will be replaced with the object from the right-hand side of the assignment statement.

   This means that the same name may exist in many different namespaces but Python will always use the name as governed by rule 1.

To show these rules in action, consider the code shown in Session 6.14. Here, function `test1` defines one formal parameter value called `a`. It adds 5 to `a` and prints the result. Since `a` is a formal parameter, it becomes part of the local namespace for the function `test1`.

Next, an assignment statement creates a variable called `a` and sets it to refer to the object 6. This occurrence of the name `a` is added to the main namespace. When we invoke `test1` using `a` as the actual parameter, call by assignment parameter passing will first evaluate `a`. The result is a reference to the object 6, which is passed and received by the formal parameter `a` in the local namespace. When the assignment statement is performed in the function `test1`, Python must search for the name `a`. The result of the search is that the `a` from the local namespace is used in the statement `a = a + 5`. The `a` in the main namespace is unaffected and still refers to the object 6.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

```
>>> def test1(a):
...     a = a + 5
...     print(a)
...
>>> a = 6
>>> test1(a)
11
>>> a
6

>>> def test2(b):
...     print(b)
...     print(a)
...
>>> test2(14)
14
6
>>> a
6
>>> b
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'b' is not defined
>>>
```

Session 6.14 Showing `name` lookup rules at work

In the second example in Session 6.14, `test2` is defined with a formal parameter called `b`. This function prints `b` and then prints `a`. However, the name `a` is not defined in the local namespace for `test2`. When the print statements are executed, we will use the previous rules to locate the names. The result of the search is that `b` is found in the local namespace but `a` is found in the main namespace.

In this example, the reference to the object 14 is assigned to the formal parameter `b` in `test2` so that the first `print` statement simply prints the value 14. The second `print` statement tries to find a variable called `a`. Since it cannot be found in the local scope, the search proceeds outward to the main namespace where `a` is found with a value of 6. Therefore, 6 is printed.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

Note that after the call to `test2` completes, `a` still has the value 6 since it is part of the main namespace. Once the function `test2` completes, the namespace is destroyed and `b` is no longer present. Since `b` does not exist in the main namespace, an error is reported that `b` is not defined.

6.4.4 Modules and Namespaces

The `hypotenuse` function defined earlier uses the `sqrt` function from the math library. In order to access that function, we needed to import the math module. In Python, the statement `import math` creates a name in the current namespace with a reference to a new namespace for the module itself. In this case, the name `math` is added to the main namespace. The new namespace for the math module is placed in the built-in namespace, as is shown in Figure 6.12. The math namespace contains names of functions such as `sqrt`. Note that the name `sqrt` refers to a function definition.

It is important to note that the namespace for the math module was placed in the built-in namespace, not in the main namespace. The namespaces for all imported modules will be placed at the same level within the built-in namespace. The only thing that will be placed

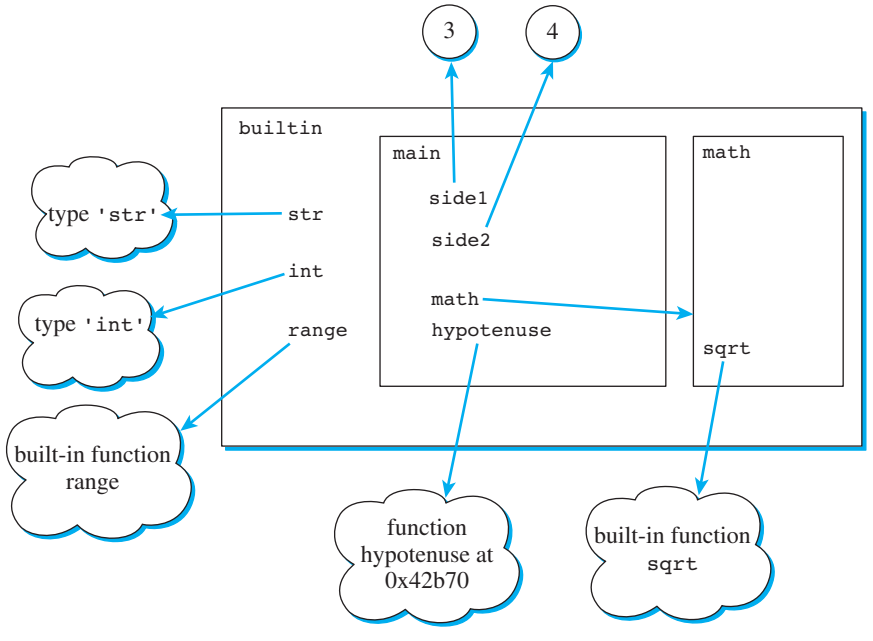

Figure 6.12 Namespaces including imported module

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

directly in the main namespace is the name of the module with the corresponding reference to the module namespace.

To take this one step further, consider the `hypotenuse` function at the point where the `sqrt` function has been invoked. As before, the namespace for `hypotenuse` has been placed as a local namespace in the main namespace. When the `hypotenuse` function calls the `sqrt` function, a new local namespace for `sqrt` is created. Even though the call to `sqrt` was made from the `hypotenuse` namespace, the `sqrt` namespace is placed in the math namespace since that is where the `sqrt` function was defined.

The local namespace created when a function is called is always created in the namespace for the module in which the function was defined. In our example, this is true regardless of how the `math` module is imported. Even if we had imported `math` using `from math import *` the namespace for `sqrt` would be placed in the math namespace.

Figure 6.13 shows all of the namespaces up to this point. According to the name lookup rules defined earlier, searches for names used in the `sqrt` function will start in the local `sqrt` namespace, proceed outward to the math namespace, and finally to the built-in namespace.

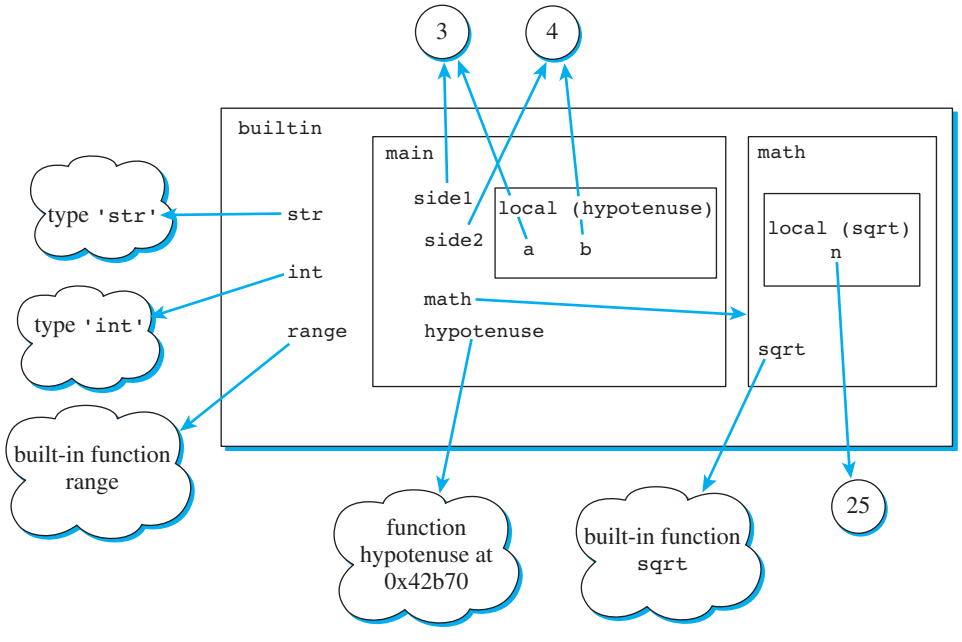

Figure 6.13 Invocation of `sqrt` function from `math`

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

6.5 Advanced Image Processing

We now turn our attention to some image processing algorithms that require the manipulation of more than one pixel either in the original or in the new image. These techniques will require that we look for additional patterns in the way that we process the pixels of the image.

6.5.1 Resizing

One of the most common manipulations performed on images is **resizing**—the process of increasing or decreasing the dimensions (width and height) of an image. In this section we focus on enlarging an image. In particular, we consider the process of creating a new image that is twice the size of the original.

Figure 6.14 shows the basic idea. The original image is three pixels wide by four pixels high. When we enlarge the image by a factor of 2, the new image will be 6 pixels wide by 8 pixels high. This presents a problem with respect to the individual pixels within the image.

The original image has 12 individual pixels. No matter what we do, we will not be able to create any new detail in the image. This means that when we increase the number of pixels to 48 in the new image, 36 of the pixels must use information that is already present in the original. Our problem is to decide systematically how to "spread" the original detail over the pixels of the new image.

Figure 6.15 shows one possible solution to this problem. Each pixel of the original will be mapped into 4 pixels in the new image. Every 1-by-1 block of pixels in the original image

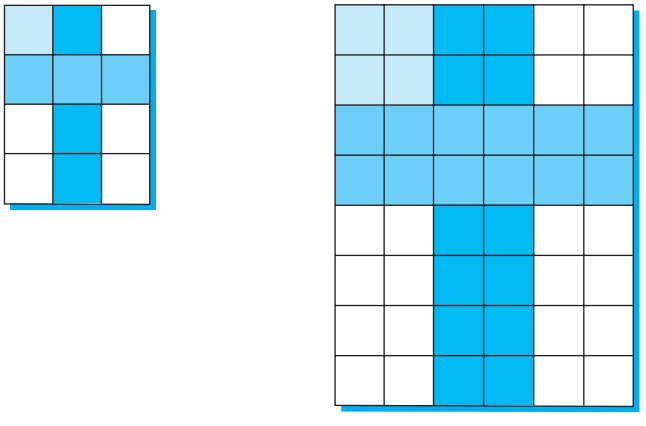

Figure 6.14 Enlarging an image by a factor of 2

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

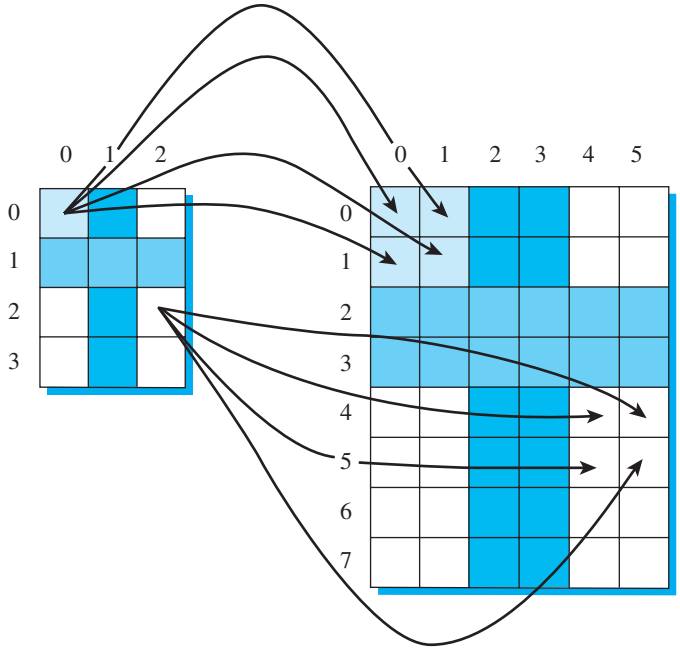

Figure 6.15 Mapping each old pixel to 4 new pixels

is mapped into a 2-by-2 block in the new image. This results in a one-to-four mapping that will be carried out for all of the original pixels. Our task is to discover a pattern for mapping a pixel from the original image onto the new image.

An example of this mapping process shows that pixel $(0, 0)$ will be mapped to pixels $(0, 0)$, $(1, 0)$, $(0, 1)$, $(1, 1)$. Likewise, pixel $(2, 2)$ maps to pixels $(4, 4)$, $(5, 4)$, $(4, 5)$, $(5, 5)$. Extending this pattern to the general case of a pixel with location $(\texttt{col}, \texttt{row})$ gives the four pixels `(2 × col, 2 × row)`, `(2 × col + 1, 2 × row)`, `(2 × col, 2 × row + 1)`, `(2 × col + 1, 2 × row + 1)`. Figure 6.16 illustrates the mapping equations for a particular pixel. You should check your understanding by considering other pixels in the original image.

Listing 6.8 shows the complete function for doubling the size of an image. Since the new image will be twice the size of the old, it will be necessary to create an empty image with dimensions that are double those of the original (see lines 2–5).

Now we can use nested iteration to process each original pixel. As before, two *for* loops, one for the columns and one for the rows, will allow us to systematically process each pixel. Using the color components from each old pixel, we copy them to the new image. Lines 11–14 use the pattern discussed above to assign each pixel in the new image. Note that each of the four pixels receives the same color tuple. The result can be seen in Figure 6.17.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

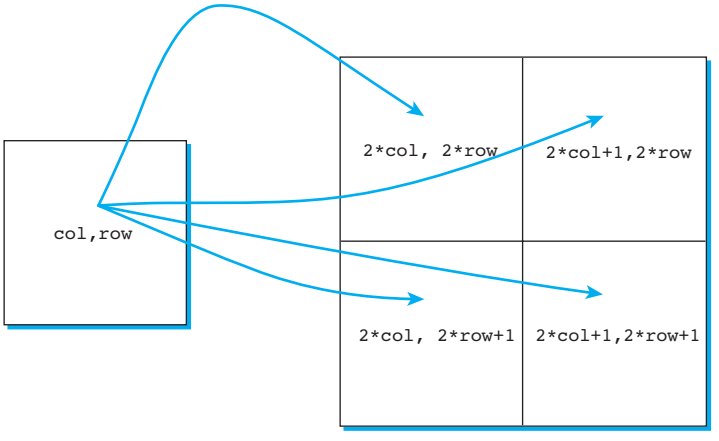

Figure 6.16 Mapping for a pixel at location (`col`, `row`)

```
def double(oldimage):
    oldw = oldimage.getWidth()
    oldh = oldimage.getHeight()

    newim = EmptyImage(oldw*2,oldh*2)

    for row in range(oldh):
        for col in range(oldw):
            oldpixel = oldimage.getPixel(col,row)

            newim.setPixel(2*col,2*row,oldpixel)
            newim.setPixel(2*col+1,2*row,oldpixel)
            newim.setPixel(2*col,2*row+1,oldpixel)
            newim.setPixel(2*col+1,2*row+1,oldpixel)

    return newim
```

Listing 6.8 Doubling the size of an image

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

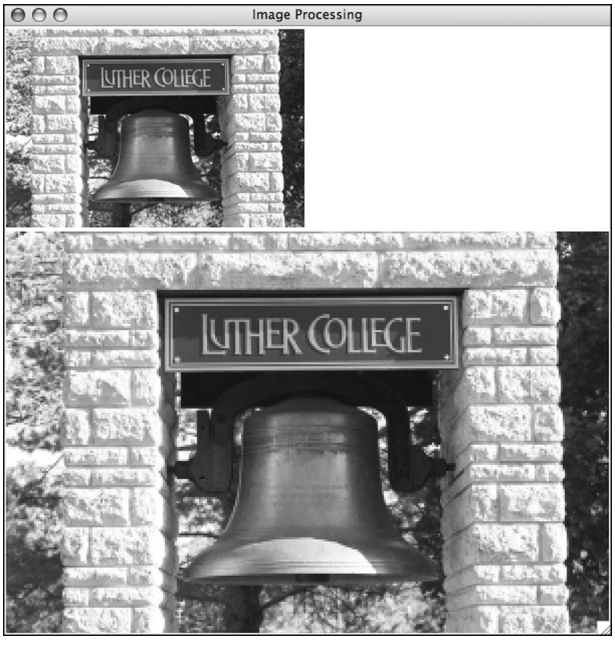

Figure 6.17 Enlarged image

6.5.2 Stretching: A Different Perspective

In the previous section we developed an algorithm for enlarging an image. That solution required us to map each pixel in the original image to 4 pixels in the new image. In this section we consider an alternative solution: constructing a new image by mapping pixels from the new image to the original. Viewing problems from many different perspectives can often provide valuable insight. Our alternative solution takes advantage of this insight and leads to a simpler solution.

Figure 6.18 shows the same image as before. However, this time the pixel mapping is drawn in the reverse direction. More specifically, instead of looking at the problem from the perspective of the original image, we are turning our focus to the pixels of the new image. As we process the pixels of the new image, we need to figure out which pixel in the original image should be used.

Listing 6.9 shows the completed code for our new function, which will take an original image as a parameter and return the new, enlarged image. Again, we will need a new empty image that is twice the size of the original. This time we write our iteration to process each pixel in the new image. The nested iteration idea will still work but the bounds will need to be in terms of the new image, as can be seen in lines 7–8.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

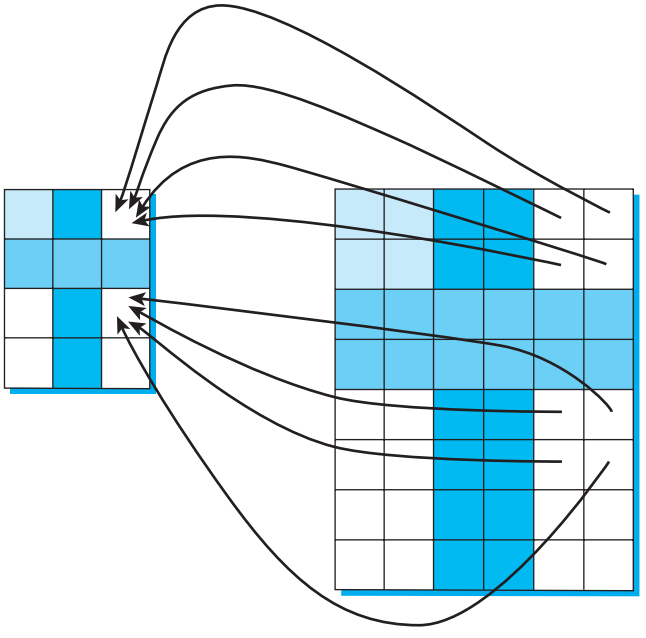

Figure 6.18 Mapping each of 4 new pixels back to 1 old pixel

```
def double(oldimage):
    oldw = oldimage.getWidth()
    oldh = oldimage.getHeight()

    newim = EmptyImage(oldw*2,oldh*2)

    for row in range(newim.getHeight()):
        for col in range(newim.getWidth()):

            originalCol = col//2
            originalRow = row//2
            oldpixel = oldimage.getPixel(originalCol,originalRow)

            newim.setPixel(col,row,oldpixel)

    return newim
```

Listing 6.9 Doubling the size of an image: Mapping new back to old

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

We now need to perform the pixel mapping. As we saw in the last section, the pixels at locations (4, 4), (5, 4), (4, 5), (5, 5) will all map back to pixel (2, 2) in the original. As another example (see again Figure 6.18, pixels (4, 0), (5, 0), (4, 1), (5, 1) will all map back to pixel (2, 0)). Our task is to find the mapping pattern that will allow us to locate the appropriate pixel in the general case.

Once again, it may appear that there are four cases since four pixels in the new image associate to a single pixel in the original. However, upon further examination, that is not true. Since we are considering the problem from the perspective of the new image, there is only `one` pixel that is of interest in the original. This suggests that there may be a single operation that will map each of the new pixels back to the original. Looking at the example pixels, we can see that integer division will perform the operation that we need (see Figure 6.19).

More specifically, in the examples given above, we need an operation that can be done to both 4 and 5 where the result will be 2. Also, the same operation on 0 and 1 will need to yield 0. Recall that both `4//2` and `5//2` give a result of 2 since the // operator when working on integers gives an integer result while discarding the remainder. Likewise, `0//2` and `1//2` both give 0 as their result.

We can now use this operation to complete the function. Lines 10–11 extract the correct pixel from the original image by using the integer division operator to compute the corresponding column and row. Once the pixel has been chosen, we can assign it to the location in the new image. Of course, the result is the same as seen in the last section (see again Figure 6.17).

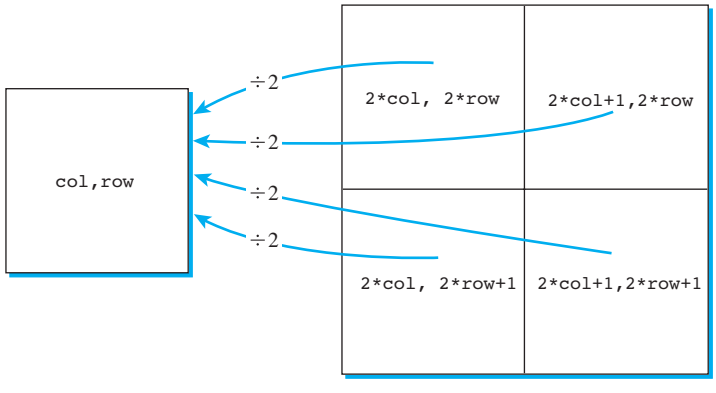

Figure 6.19 Mapping back using integer division

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

As we stated earlier, enlarging an image provides no new detail. The result can therefore look "grainy" or "blocky" due to the fact that we are mapping one pixel to many locations in the new image. Although we cannot create any new detail to add to the image, it is possible to "smooth" out some of the hard edges by processing each pixel with respect to those around it.

Exercises

6.22 Write a function to quadruple the size of an image.

6.23 Write a general function for enlarging an image that accepts a scale parameter for enlarging in the x direction and another parameter for enlarging in the y direction.

6.24 Write a function for reducing the size of an image.

6.25 Write a function that will smooth the enlarged image. *Hint:* You will want to replace each pixel in the enlarged image with the average of itself and its neighbors.

6.26 Write a function to remove noise from an image. You can do this by replacing each pixel with the median of itself and its neighbors.

6.5.3 Flipping an Image

We now consider manipulations that physically transform an image by moving pixels to new locations. In particular, we consider a process known as **flipping**. Creating a **flip image** requires that we decide where the flip will occur. For our purposes in this section, we will assume that flipping happens with respect to a line that we will call the **flip axis**. The basic idea is to place pixels that appear on one side of the flip axis in a new location on the opposite side of that axis, keeping the distance from the axis the same.

As an example, consider the simple image with 16 pixels in Figure 6.20 and a flip axis placed vertically at the center of the image. Because we are flipping on the vertical axis, each row will maintain its position relative to every other row. However, within each row, the pixels will move to the opposite side of the axis, as shown by the arrows. The first pixel will be last and the last pixel will be first.

The structure of this function is similar to those we have written thus far. We build our nested iteration such that the outer iteration will process the rows and the inner iteration will process each pixel within the row. Listing 6.10 shows the completed function. Note that the new image is the same height and width as the original.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

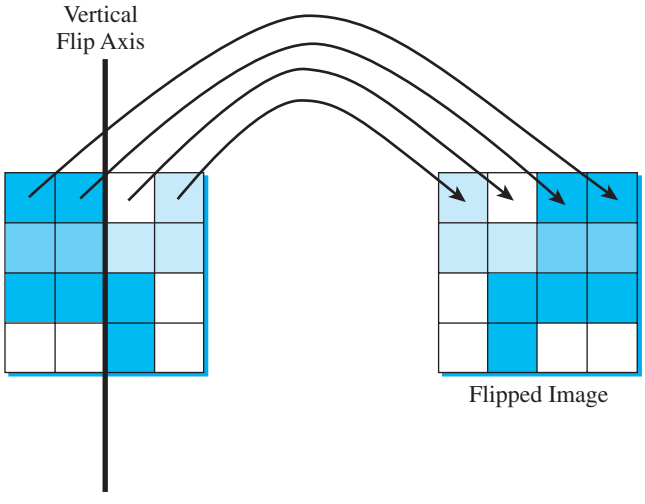

Figure 6.20 Flipping an image on the vertical axis

```
def verticalFlip(oldimage):
    oldw = oldimage.getWidth()
    oldh = oldimage.getHeight()

    newim = EmptyImage(oldw,oldh)

    maxp = oldw-1
    for row in range(oldh):
        for col in range(oldw):

            oldpixel = oldimage.getPixel(maxp-col,row)

            newim.setPixel(col,row,oldpixel)

    return newim
```

Listing 6.10 Creating the vertical flip of an image

We need to discover a pattern to map each pixel from its original location into a new location with respect to the flip axis. Referring to Figure 6.20 we can see that the following associations are needed in an image that is 4 pixels wide: column 0 will map to column 3, column 1 will map to column 2, column 2 will map to column 1, and column 3 will map to column 0. In general, small values map to large values and large values map to small.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

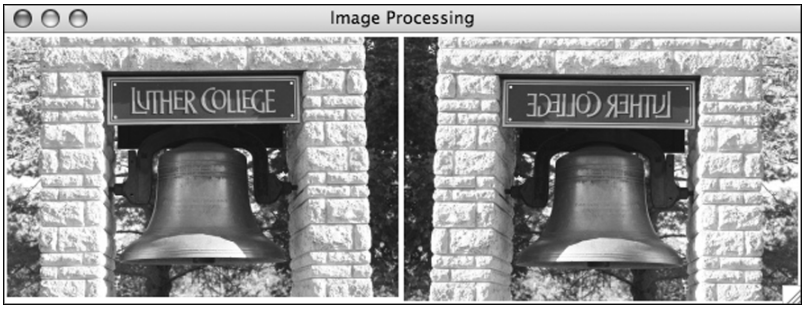

Figure 6.21 A flipped image

The first thing we might try is to use the width and simply subtract the original column to get the new column. If we try this with column 0, we immediately see that there is a problem since $4 - 0$ is 4, which is outside the range of valid column values. The cause of this error is that we start counting the columns (as well as the rows) with zero.

To fix this, we can base the subtraction on the actual maximum pixel position instead of the width. Since the pixels in this example are named with column 0 though column 3 (width of 4), we can use 3 as our base for the subtraction. In this case, the general mapping equation will be `(width-1) - column`. Note that `width-1` is a constant, which means that we can perform this calculation just once, outside the loop as we do on line 7.

Since we are performing a flip using a vertical flip axis, the pixels stay in the same row. Line 11 uses the calculation above to extract the proper pixel from the original image and line 13 places it in its new position in the new image. Note that `row` is used in both `getPixel` and `setPixel`. Figure 6.21 shows the resulting image.

Exercises

6.27 Write the function `horizontalFlip` to flip an image on the horizontal axis.

6.28 Rewrite the `verticalFlip` function so that it flips an image in place.

6.29 *Mirroring* is a manipulation similar to flipping. When producing a mirror, the pixels on one side of the mirror axis are reflected back on the other side. In a mirror operation half of the pixels are lost. Implement a mirror on the vertical axis.

6.30 Implement a mirror on the horizontal axis.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

6.31 Implement a mirror at a specific column or row. *Note:* This operation will change the image size.

6.32 Write a function `rotateImage90` that takes an image as a parameter and rotates the image by 90 degrees.

6.33 Write a function `rotateImage180` that takes an image as a parameter and rotates it by 180 degrees.

6.34 Write a function `rotate` that takes an image and a number of degrees to rotate the image. Note that this rotation may leave some empty pixels. You will also need to size the new image so it can hold the entire rotated image.

6.5.4 Edge Detection

Our final image processing algorithm in this chapter is called **edge detection**—an image processing technique that tries to extract feature information from an image by finding places in the image that have very dramatic changes in color intensity values. For example, assume that we have an image containing two apples, one red and one green, that are placed next to one another. The border between a block of red pixels from the red apple and a block of green pixels from the green apple might constitute an edge representing the distinction between the two objects.

As another example, consider the grayscale image (actually black and white) shown in Figure 6.22. In the left image there are three objects: a white square, a cloud, and a star. The right image shows the edges that exist. Each black pixel in the edge image denotes a point where there is a distinct difference in the intensity of the original grayscale pixels. Finding these edges helps to differentiate between any features that may exist in the original image.

Edge detection has been studied in great detail. There are many different approaches that can be used to find edges within an image. In this section we describe one of the classic algorithms for producing the edges. The mathematics used to derive the algorithm are beyond the scope of this book. However, we can easily develop the ideas and techniques necessary to implement the algorithm.

In order to find an edge, it is necessary to evaluate each pixel in relation to those that appear around it. Since we are looking for places where intensity on one side of the pixel is greatly different from the intensity on the other, it will help us to simplify the pixel values. Our first step in discovering edges will be to convert the image to grayscale. This means that the intensity of the pixel can be thought of as the common color component intensity.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

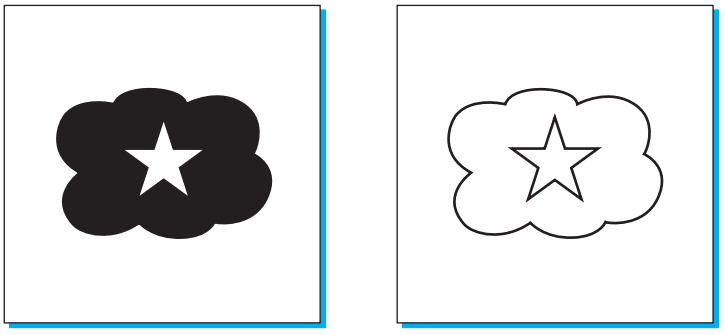

Figure 6.22 A simple edge detection

(Recall that shades of gray are made from pixels with equal quantities of red, green, and blue.) Each pixel can then be thought to have one of 256 intensity values.

As a means of looking for these intensity differences, we use the idea of a **kernel**, also known as a *filter* or a *mask*. These kernels will be used to weight the intensities of the surrounding pixels. For example, consider the 3 by 3 kernels shown in Figure 6.23. These "grids" of integer weights are known as the **Sobel operators**, named after Irwin Sobel who developed them for use in edge detection.

The left mask, labeled `XMask`, will be used to look for intensity differences in the left to right direction of the image. You can see that the leftmost column of values is negative and the rightmost column is positive. Likewise, the `YMask` will look for intensity differences in the up and down direction as can be seen by the location of the positive and negative weights.

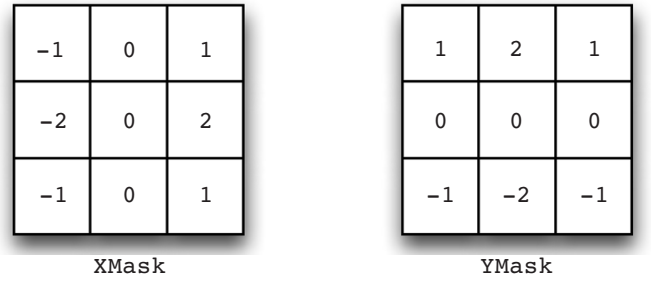

Figure 6.23 Kernel masks for convolving pixels

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

The kernels will be used during **convolution**—a process in which pixels in the original image will be mathematically combined with each mask. The result will then be used to decide whether that pixel represents an edge.

Convolution will require a mask and a specific pixel. The mask will be "centered" on the pixel of interest, as shown in Figure 6.24. Each weight value in the mask now associates with one of the nine pixel intensities "under" the mask. The convolution process will simply compute the sum of nine products where each product is the weight multiplied by the intensity of the associated pixel. If you have a large intensity on the left side and a small intensity on the right (indicating an edge), you will get a weighted sum with a large negative value. If you have a small intensity on the left and a large intensity on the right, you will get a large positive weighted sum. Either way, a large absolute value of the weighted sum will indicate an edge. This same argument applies for the top-to-bottom split of the `YMask`.

To implement convolution, we will first need to consider a way to store the kernels. Since kernels look very similar to images, rows and columns of weights, it makes sense to take advantage of this structure. We will use a **list of lists** implementation for the kernel. For example, the `XMask` discussed earlier will be implemented as the list `[ [-1,0,1],[-2,0,2],[-1,0,1] ]`. The outer list contains three items, each of which represents a row in the kernel. Each row has three items, one for each column. Similarly, the `YMask` will be `[ [1,2,1],[0,0,0],[-1,-2,-1] ]`.

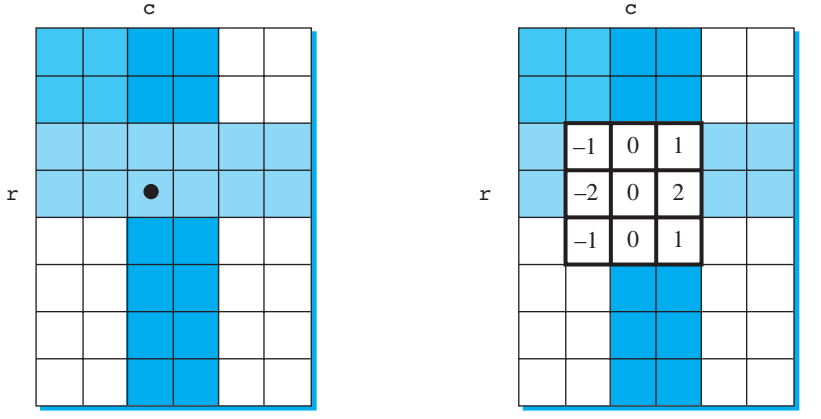

Figure 6.24 Using the `XMask` to convolve the pixel at `c,r`

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

Accessing a specific weight within a kernel will require two index values, one for the outer list and one for the inner list. Since we are implementing the outer list to be a list of three rows, the first index will be the row value. Once we select a row list, the second index will be used to get the specific column.

For example, `XMask[1][2]` will access the item in `XMask` indexed by 1, which is the middle row of the `XMask`. The `2` indexes the last item in the list that corresponds to the last column. This access is for the weight stored in the middle row and last column of `XMask`.

We can now construct the `convolve` function. We said earlier that this process requires an image, a specific pixel within the image, and a kernel. The tricky part of this function is to align the kernel and the underlying image. An easy way to do this is to think about a mapping. The kernel row indices will run from 0 to 2. Likewise for the column indices. For a pixel in the image with index `(column,row)`, the row indices for the underlying pixels will run from `row-1` to `row+1` and for the columns it will be `column-1` to `column+1`.

We will define the *base index* to be the starting index for the 3×3 grid of underlying image pixels. The base index for the columns will be `column-1` and the base index for the rows will be `row-1`. As we process the pixels of the image, the difference between the current image row value and the base index for the rows will be equal to the row index needed to access the correct row in the kernel. Likewise, we can do the same thing for the columns.

Once we have computed the index into the kernel we can use it to compute the product of the weight and the pixel intensity. We will first access the pixel and then extract the red component that will be its grayscale intensity. Since we have already converted the image to grayscale, we can use any one of the red, green, or blue components for the intensity. Finally, that product can be added to a running sum of products for all underlying pixels. The complete `convolve` function is shown in Listing 6.11. Note that the final step is to return the value of the sum.

Now that we can perform the convolution operation for a specific pixel with a kernel, we can complete the edge detection algorithm. The steps of the process are as follows:

1. Convert the original image to grayscale.
2. Create an empty image with the same dimension as the original.
3. Process each inner pixel of the original image by performing the following:
   (a) Convolve the pixel with the `XMask`; call the result `gX`.
   (b) Convolve the pixel with the `YMask`; call the result `gY`.
   (c) Compute the square root of the sum of squares of `gX` and `gY`; call the result `g`.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

```
def convolve(anImage,pixelRow,pixelCol,kernel):
    kernelColumnBase = pixelCol - 1
    kernelRowBase = pixelRow - 1

    sum = 0
    for row in range(kernelRowBase,kernelRowBase+3):
        for col in range(kernelColumnBase,kernelColumnBase+3):
            kColIndex = col-kernelColumnBase
            kRowIndex = row-kernelRowBase

            apixel = anImage.getPixel(col,row)
            intensity = apixel.getRed()

            sum = sum + intensity * kernel[kRowIndex][kColIndex]

    return sum
```

Listing 6.11 Convolution for a specific pixel

(d) Based on the value of g, assign the corresponding pixel in the new image to be either black or white.

Listing 6.12 shows the Python code that implements the steps outlined previously. We begin by converting the original image to grayscale using the `pixelMapper()` function developed earlier in the chapter. This will allow for simple intensity levels within each pixel. We will also need an empty image that is the same size as the original. It will also be useful to define a few data objects for use later. Since each pixel in the edge detection result will be either black or white, we will create black and white tuples that can be assigned later in the process. Also, we will need the list of lists implementation of the two kernels. These initializations are done on lines 3–8.

Now it is time to process the original pixels looking for an edge. Since each pixel is required to have eight surrounding pixels for the convolution operation, we will not process the first and last pixel on each row and column. This means that our nested iteration will start at one, not zero, and it will continue through `height-2` and `width-2` as shown on lines 10–11.

Each pixel will now participate in the convolution process using both kernels. The resulting sums will be squared and summed together, and in the final step we will take the square root (see lines 12–14).

The value of this square root, called g, represents a measure of how much difference exists between the pixel and those around it. The decision as to whether the pixel should be

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

```
import math
def edgeDetect(theImage):
    grayImage = pixelMapper(theImage,grayPixel)
    newim = EmptyImage(grayImage.getWidth(), grayImage.getHeight())
    black = Pixel(0,0,0)
    white = Pixel(255,255,255)
    XMask = [ [-1,-2,-1],[0,0,0],[1,2,1] ]
    YMask = [ [1,0,-1],[2,0,-2],[1,0,-1] ]

    for row in range(1,grayImage.getHeight()-1):
        for col in range(1,grayImage.getWidth()-1):
            gx = convolve(grayImage,row,col,XMask)
            gy = convolve(grayImage,row,col,YMask)
            g = math.sqrt(gx**2 + gy**2)

            if g > 175:
                newim.setPixel(col,row,black)
            else:
                newim.setPixel(col,row,white)

    return newim
```

Listing 6.12 The edge detection method

labeled as an edge is made by comparing g to a threshold value. It turns out that when using these kernels, 175 is a good threshold value for considering whether you have found an edge. Using simple selection, we will just check to see if the value is greater than 175. If it is, we will color the pixel black; otherwise we will make it white. Figure 6.25 shows the result of executing this function.

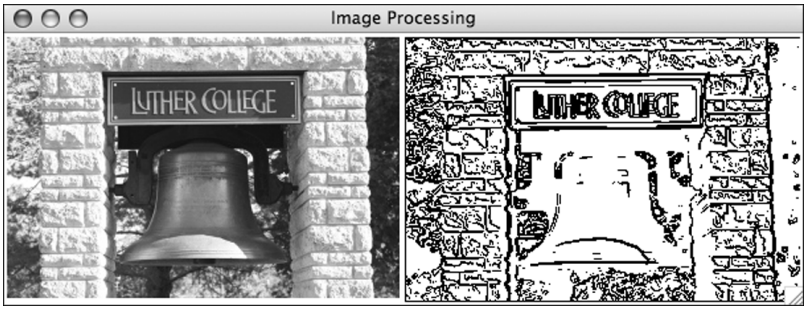

Figure 6.25 Running the edge detection algorithm

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

Exercises

6.35 Try several different threshold values in `edgeDetect`. What effect does changing the threshold have on the image? Does 175 work best for all images? What would be a way to automatically select a good threshold for an image?

6.36 Modify the convolve function so that it applies the kernel to the red, green, and blue components separately and returns a tuple of values as a result.

6.37 Convolution has many uses. For example, a simple convolution kernel is the blurring kernel, which looks like this:

$$\begin{bmatrix} 1 & 2 & 1 \\ 2 & 1 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

In this case we simply apply the kernel and return the weighted average without doing any thresholding. Write a blur function that uses the new convolve function to blur an image.

6.38 The sharpen kernel looks like this:

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

You can sharpen a pixel by emphasizing its value and deemphasizing the pixels around it. Sharpening is the opposite of blurring. Use the sharpen kernel to sharpen an image.

6.39 Write a general function that can take an image and a kernel and then return an image with the convolution kernel applied to each pixel.

6.40 Research convolution kernels and find a new one to try.

6.6 Summary

In this chapter we focused on `cImage`—a new module that contains a number of data types that can be used to manipulate digital images. In particular, `cImage` includes the following:

- `ImageWin`
- `EmptyImage`

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

- `FileImage`
- `Pixel`

To process the pixels of an image, we used a pattern called nested iteration—that is, iteration inside iteration. Nested iteration allowed us to process all of the pixels in a given row, column by column, before moving on to the next row. We also introduced the notion of namespaces—collections of names available at a particular point in time. These namespaces are organized to allow us to look up names when they are used, thereby making sure that there are no ambiguities. This chapter concluded with a more detailed consideration of the mechanics of parameter passing.

Key Terms

actual parameter	flip image	list of lists	pixel
built-in namespace	flipping	local namespace	Pythagorean theorem
call by assignment	formal parameter	main namespace	resizing
convolution	gray scale	namespace	RGB color model
digital image	image processing	negative	Sobel operators
edge detection	invocation	nested iteration	
flip axis	kernel	parameter passing	

Python Keywords

`dir`	`for`	`math`	`return`
`EmptyImage`	`ImageWin`	`Pixel`	
`FileImage`	`import`	`range`	

Bibliography

[Par96] J. R. Parker. *Algorithms for Image Processing and Computer Vision.* Wiley, 1996.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

Programming Exercises

6.1 Write a program to create a collage. Your program should combine several images with different effects applied to the images.

6.2 Write a program to blend one image with another. You can try different techniques for combining the RGB values for two pixels, each from a different image.

6.3 Take a picture of yourself against a white background. Use the fact that you can "filter" out all the white pixels to place your picture in an interesting scene. This same process is used all the time by weather forecasters on television. The only difference is that they stand in front of a solid blue or solid green background called a *chromakey*.

6.4 Another way to put yourself in an interesting picture is to take a picture of yourself against a relatively plain background, then take another picture of exactly the same background (use a tripod here with autofocus off) without you in it. Now you can compare the two images and remove the pixels that are exactly the same, or close to the same. Once you have removed those pixels, you can superimpose yourself on any background.

6.5 Using `getMouse` to get the coordinates of a pixel in an image, devise a way to remove the red-eye effect from the area of the image you click on.

6.6 Using `getMouse`, write a program that will allow you to "cut" a rectangular region out of an image and place it somewhere in a new image.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.

© Jones & Bartlett Learning, LLC. NOT FOR SALE OR DISTRIBUTION.