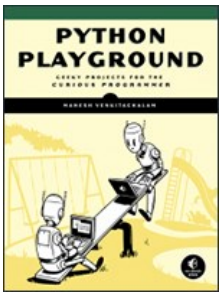

Python Playground: Geeky Projects for the Curious Programmer

by Mahesh Venkitachalam

Chapter 7: Photomosaics

Overview

When I was in the sixth grade, I saw a picture like the one shown in Figure 7-1 but couldn't quite figure out what it was. After squinting at it for a while, I eventually figured it out. (Turn the book upside down, and view it from across the room. I won't tell anyone.)

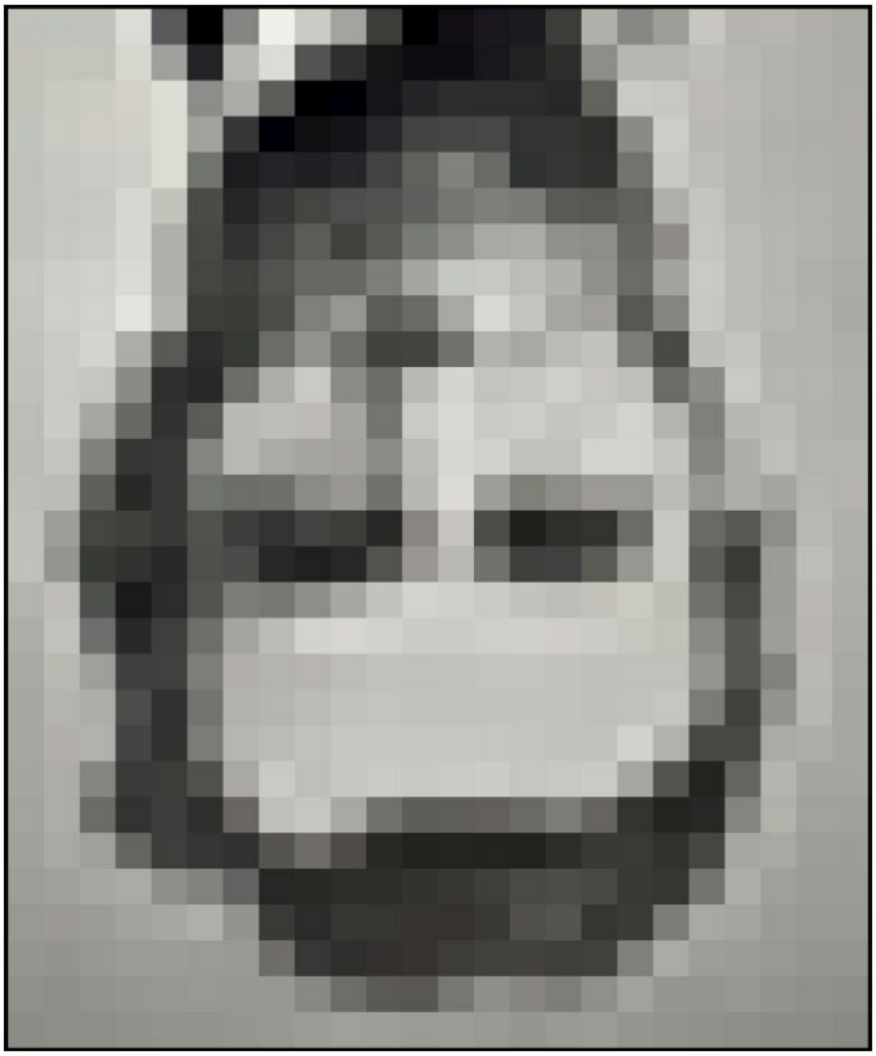

Figure 7-1: A puzzling image

A *photomosaic* is an image split into a grid of rectangles, with each replaced by another image that matches the *target* (the image you ultimately want to appear in the photomosaic). In other words, if you look at a photomosaic from a distance, you see the target image; but if you come closer, you will see that the image actually consists of many smaller images.

The puzzle works because of how the human eye functions. The lowresolution, blocky image shown in Figure 7-1 is hard to recognize up close, but when seen from a distance, you know what it represents because you perceive less detail, which makes the edges smooth. A photomosaic works according to a similar principle. From a distance, the image looks normal, but up close, the secret is revealed—each "block" is a unique image!

In this project, you'll learn how to create photomosaics using Python. You'll divide a target image into a grid of smaller images and replace each block in the grid with a suitable image to create a photomosaic of the original image. You'll be able to specify the grid dimensions and choose whether input images can be reused in the mosaic.

In this project, you'll learn how to do the following:

- Create images using the Python Imaging Library (PIL).

- Compute the average RGB value of an image.
- Crop images.
- Replace part of an image by pasting in another image.
- Compare RGB values using a measurement of average distance.

How It Works

To create a photomosaic, begin with a blocky, low-resolution version of the target image (because the number of tile images would be too great in a high-resolution image). The resolution of this image will determine the dimensions *M* × *N* (where *N* is the number of rows and *N* is the number of columns) of the mosaic. Next, replace each tile in the original image according to this methodology:

1. Read the tile images, which will replace the tiles in the original image.
2. Read the target image and split it into an *M* × *N* grid of tiles.
3. For each tile, find the best match from the input images.
4. Create the final mosaic by arranging the selected input images in an *M* × *N* grid.

Splitting the Target Image

Begin by splitting the target image into an *M* × *N* grid according to the scheme shown in Figure 7-2.

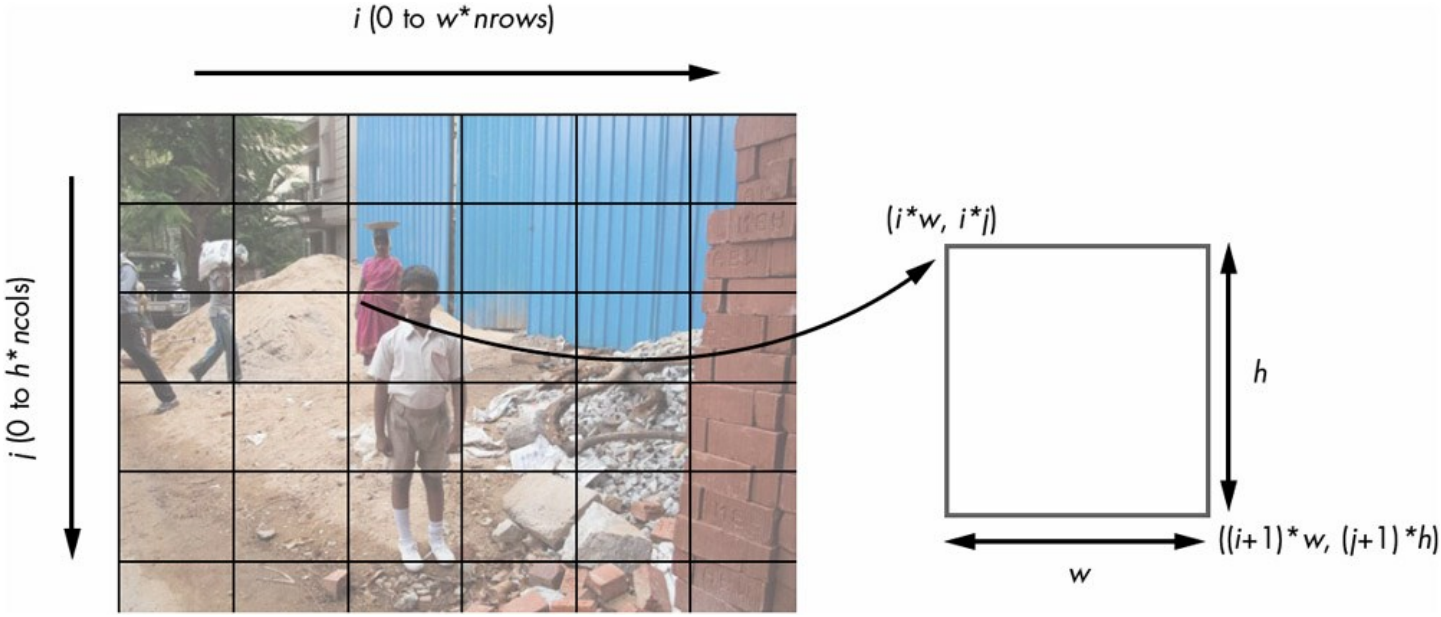

Figure 7-2: Splitting the target image

The image in Figure 7-2 shows how you can split the original image into a grid of tiles. The x-axis represents the grid columns, and the y-axis represents the grid rows.

Now let's look at how to calculate the coordinates for a single tile from this grid. The tile with index (*i*, *j*) has a top-left corner coordinate of (*i***w*, *i***j*) and a bottom-right corner coordinate of ((*i*+1)**w*, (*j*+1)**h*), where *w* and *h* stand for the width and height of a tile, respectively. These can be used with the PIL to crop and create a tile from this image.

Averaging Color Values

Every pixel in an image has a color that can be represented by its red, green, and blue values. In this case, you are using 8-bit images, so each of these components has an 8-bit value in the range [0, 255]. Given an image with a total of *N* pixels, the average RGB is calculated as follows:

$$(r,g,b)_{\text{avg}} = \left(\frac{(r_1 + r_2 + \cdots + r_N)}{N}, \frac{(g_1 + g_2 + \cdots + g_N)}{N}, \frac{(b_1 + b_2 + \cdots + b_N)}{N} \right)$$

Note that the average RGB is also a triplet, not a scalar or single number, because the averages are calculated separately for each color component. You calculate the average RGB to match the tiles with the target image.

Matching Images

For each tile in the target image, you need to find a matching image from the images in the input folder specified by the user. To determine whether two images match, use the average RGB values. The closest match is the image with the closest average RGB value.

The simplest way to do this is to calculate the distance between the RGB values in a pixel to find the best match among the input images. You can use the following distance calculation for 3D points from geometry:

$$D_{1,2} = \sqrt{(r_1 - r_2)^2 + (g_1 - g_2)^2 + (b_1 - b_2)^2}$$

Here you compute the distance between the points (r_1, g_1, b_1) and (r_2, g_2, b_2). Given a target average RGB value and a list of average RGB values from the input images, you can use a linear search and the distance calculation for 3D points to find the closest matching image.

Requirements

For this project, you'll use Pillow to read in the images, access their underlying data, and create and modify the images. You'll also use numpy to manipulate image data.

The Code

You'll begin by reading in the tile images that you'll use to create the photomosaic. Next, you'll compute the average RGB value of the images, and then split the target into a grid of images and find the best matches for tiles. Finally, you'll assemble the image tiles to create the final photomosaic. To see the complete project code, skip ahead to "The Complete Code" on page 110.

Reading in the Tile Images

First, read in the input images from the given folder. Here's how to do that:

```
 def getImages(imageDir):
     """
     given a directory of images, return a list of Images
     """
1    files = os.listdir(imageDir)
     images = []
     for file in files:
2        filePath = os.path.abspath(os.path.join(imageDir, file))
         try:
             # explicit load so we don't run into resource crunch
3            fp = open(filePath, "rb")
             im = Image.open(fp)
             images.append(im)
             # force loading the image data from file
4            im.load()
             # close the file
5
            fp.close()
         except:
             # skip
             print("Invalid image: %s" % (filePath,))
     return images
```

At 1, you use os.listdir() to gather the files in the *imageDir* directory in a list. Next, you iterate through each file in the list and

load it into a PIL Image object.

At 2, you use os.path.abspath() and os.path.join() to get the complete filename of the image. This idiom is commonly used in Python to ensure that your code will work with both relative paths (for example, *\foo\bar*) and absolute paths (*c:\foo\bar*), as well as across operating systems with different directory-naming conventions (\ in Windows versus / in Linux).

To load the files into PIL Image objects, you could pass each filename to the Image.open() method, but if your photo mosaic folder had hundreds or even thousands of images, doing so would be highly resource intensive. Instead, you can use Python to open each tile image and pass the file handle *fp* into PIL using Image.open(). Once the image has been loaded, close the file handle and release the system resources.

At 3, you open the image file using open(). In the lines that follow, you pass the handle to Image.open() and store the resulting image, im, in an array.

You call Image.load() at 4 to force load the image data inside im because open() is a lazy operation. It identifies the image but doesn't actually read all the image data until you try to use the image.

At 5, you close the file handle to release system resources.

Calculating the Average Color Value of the Input Images

Once you've read in the input images, you need to calculate their average color value, as well as the value for each tile in the target. Create a method getAverageRGB() to compute both values.

```
 def getAverageRGB(image):
     """
     return the average color value as (r, g, b) for each input image
     """
     # get each tile image as a numpy array
1    im = np.array(image)
     # get the shape of each input image
2    w,h,d = im.shape
     # get the average RGB value
3    return tuple(np.average(im.reshape(w*h, d), axis=0))
```

At 1, you use numpy to convert each Image object into a data array. The numpy array returned has the shape (w, h, d), where w is the weight of the image, h is the height, and d is the depth, which, in this case, is three units (one each for R, G, and B) for RGB images. You store the shape tuple at 2 and then compute the average RGB value by reshaping this array into a more convenient form with shape (w*h, d) so that you can compute the average using numpy.average() 3.

Splitting the Target Image into a Grid

Now you need to split the target image into an 1x1 grid of smaller images. Let's create a method to do that.

```
 def splitImage(image, size):
     """
     given the image and dimensions (rows, cols), return an m*n list of images
     """
1    W, H = image.size[0], image.size[1]
2    m, n = size
3    w, h = int(W/n), int(H/m)
     # image list
     imgs = []
     # generate a list of dimensions
     for j in range(m):
         for i in range(n):
             # append cropped image
4            imgs.append(image.crop((i*w, j*h, (i+1)*w, (j+1)*h)))
     return imgs
```

First, you gather the dimensions of the target image at 1 and the grid size at 2. At 3, you calculate the dimensions of each tile in the target image using basic division.

Now you need to iterate through the grid dimensions and cut out and store each tile as a separate image. At 4, image.crop()

crops out a portion of the image using the upper-left image coordinates and the dimensions of the cropped image as arguments (as discussed in "Splitting the Target Image" on page 103).

Finding the Best Match for a Tile

Now let's find the best match for a tile from the folder of input images. You create a utility method, getBestMatchIndex(), as follows:

```
def getBestMatchIndex(input_avg, avgs):
    """
    return index of the best image match based on average RGB value distance
    """

    # input image average
    avg = input_avg

    # get the closest RGB value to input, based on RGB distance
    index = 0
1   min_index = 0
2   min_dist = float("inf")
3   for val in avgs:
4   dist = ((val[0] - avg[0])*(val[0] - avg[0]) +
            (val[1] - avg[1])*(val[1] - avg[1]) +
            (val[2] - avg[2])*(val[2] - avg[2]))
5   if dist < min_dist:
        min_dist = dist
        min_index = index
    index += 1

return min_index
```

To find the best match, you compare the average RGB values of the input images. At 1 and 2, you initialize the closest match index to 0 and the minimum distance to infinity. This test will always pass the first time since any distance will be less than infinity. At 3, you loop through the values in the list of averages and start computing distances at 4 using the standard formula. (You compare squares of distance to reduce computation time.) If the computed distance is less than the stored minimum distance min_dist, it is replaced with the new minimum distance at 5. At the end of the iteration, you have the index of the average RGB value from the avgs list that is closest to input_avg. Now you can use this index to select the matching tile image from the list of tile images.

Creating an Image Grid

You need one more utility method before moving on to photomosaic creation. The createImageGrid() method will create a grid of images of size *M* × *N*. This image grid is the final photomosiac image, created from the list of selected tile images.

```
def createImageGrid(images, dims):
    """
    given a list of images and a grid size (m, n), create a grid of images
    """
1   m, n = dims

    # sanity check
    assert m*n == len(images)

    # get the maximum height and width of the images
    # don't assume they're all equal
2
   width = max([img.size[0] for img in images])
    height = max([img.size[1] for img in images])

    # create the target image
3
   grid_img = Image.new('RGB', (n*width, m*height))

    # paste the tile images into the image grid
    for index in range(len(images)):
4       row = int(index/n)
5     col = index - n*row
6
   grid_img.paste(images[index], (col*width, row*height))
```

```
    return grid_img
```

At 1, you gather the dimensions of the grid and then use assert to see whether the number of images supplied to createImageGrid() matches the grid size. (The assert method checks assumptions in your code, especially during development and testing.) Now you have a list of tile images based on the closest RGB match, which you'll use to create an image grid representing the photomosaic. Some of the selected images may not fill a tile exactly because of differences in their sizes, but that won't be a problem because you'll fill the tile with a black background first.

At 2 and in the following line, you compute the maximum width and height of the tile images. (You haven't made any assumptions regarding the size of the selected input images; the code will work whether they're the same or different.) If the input images won't completely fill a tile, the spaces between the tiles will show as the background color, which is black by default.

At 3, you create an empty Image sized to fit all images in the grid; you'll paste the tile images into this. Then you fill the image grid. At 6, you loop through the selected images and paste them into the appropriate grid using the Image.paste() method. The first argument to Image.paste() is the Image object to be pasted, and the second is the top-left coordinate. Now you need to figure out in which row and column to paste a tile image into the image grid. To do so, you express the image index in terms of rows and columns. The index of a tile in the image grid is given by *N***row* + *col*, where *N* is the number of cells along the width and (*row*, *col*) is the coordinate in this grid; at 4, you give the row from the previous formula and at 5 the column.

Creating the Photomosaic

Now that you have all the required utilities, let's write the main function that creates the photomosaic.

```
def createPhotomosaic(target_image, input_images, grid_size, reuse_images=True):
    """
    creates a photomosaic given target and input images
    """

    print('splitting input image...')
    # split the target image into tiles
1   target_images = splitImage(target_image, grid_size)

    print('finding image matches...')
    # for each tile, pick one matching input image
    output_images = []
    # for user feedback
    count = 0
2   batch_size = int(len(target_images)/10)
    # calculate the average of the input image
    avgs = []
    for img in input_images:
3       avgs.append(getAverageRGB(img))

    for img in target_images:
        # compute the average RGB value of the image
4       avg = getAverageRGB(img)
        # find the matching index of closest RGB value
        # from a list of average RGB values
5       match_index = getBestMatchIndex(avg, avgs)
6       output_images.append(input_images[match_index])
        # user feedback
7       if count > 0 and batch_size > 10 and count % batch_size is 0:
            print('processed %d of %d...' %(count, len(target_images)))
        count += 1
        # remove the selected image from input if flag set
8       if not reuse_images:
            input_images.remove(match)
    print('creating mosaic...')
    # create photomosaic image from tiles
9   mosaic_image = createImageGrid(output_images, grid_size)
    # display the mosaic
    return mosaic_image
```

The createPhotomosaic() method takes as input the target, the list of input images, the size of the generated photomosaic, and a flag that indicates whether an image can be reused. At 1, it splits the target into a grid. Once the image is split, you find matches for each tile from the images in the input folder. (Because this process can be lengthy, you provide feedback to users to let them know that the program is still working.)

At 2, you set batch_size to one-tenth the total number of tile images. This variable will be used to update the user in the code at 7. (The choice of one-tenth is arbitrary and simply a way for the program to say "I'm still alive." Each time the program processes a tenth of the images, it prints a message indicating that it's still running.)

At 3, you compute the average RGB value for each image in the input folder and store that value in the list avgs. Then you start iterating through each tile in the target image grid. For each tile, you calculate the average RGB value 4; then, at 5, you search for the closest match to this value in the list of averages for the input images. The result is returned as an index, which you use at 6 to retrieve the Image object and store it in a list.

At 7, for every batch_size number of images processed, you print a message to the user. At 8, if the reuse_images flag is set to False, you remove the selected input image from the list so that it won't be reused in another tile. (This works best when you have a wide range of input images to choose from.) Finally, at 9, you combine the images to create the final photomosaic.

Adding the Command Line Options

The main() method of the program supports these command line options:

```
# parse arguments
parser = argparse.ArgumentParser(description='Creates a photomosaic from
                                 input images')
# add arguments
parser.add_argument('--target-image', dest='target_image', required=True)
parser.add_argument('--input-folder', dest='input_folder', required=True)
parser.add_argument('--grid-size', nargs=2, dest='grid_size', required=True)
parser.add_argument('--output-file', dest='outfile', required=False)
```

This code contains three required command line parameters: the name of the target, the name of the input folder of images, and the grid size. The fourth parameter is for the optional filename. If the filename is omitted, the photomosaic will be written to a file named *mosaic.png*.

Controlling the Size of the Photomosaic

One last issue to address is the size of the photomosaic. If you were to blindly paste the input images together based on matching tiles in the target, you could end up with a huge photomosaic that is much bigger than the target. To avoid this, resize the input images to match the size of each tile in the grid. (This has the added benefit of speeding up the average RGB computation since you're using smaller images.) The main() method also handles this:

```
    print('resizing images...')
    # for given grid size, compute the maximum width and height of tiles
1   dims = (int(target_image.size[0]/grid_size[1]),
            int(target_image.size[1]/grid_size[0]))
    print("max tile dims: %s" % (dims,))
    # resize
    for img in input_images:
2       img.thumbnail(dims)
```

You compute the target dimensions at 1 based on the specified grid size; then, at 2, you use the PIL Image.thumbnail() method to resize the images to fit those dimensions.

The Complete Code

You can find the complete code for the project at *https://github.com/electronut/pp/tree/master/photomosaic/ photomosaic.py*.

```
import sys, os, random, argparse
from PIL import Image
import imghdr
import numpy as np
def getAverageRGB(image):
    """
    return the average color value as (r, g, b) for each input image
    """
    # get each tile image as a numpy array
    im = np.array(image)
```

```
    # get the shape of each input image
    w,h,d = im.shape
    # get the average RGB value
    return tuple(np.average(im.reshape(w*h, d), axis=0))
def splitImage(image, size):
    """
    given the image and dimensions (rows, cols), returns an m*n list of images
    """
    W, H = image.size[0], image.size[1]
    m, n = size
    w, h = int(W/n), int(H/m)
    # image list
    imgs = []
    # generate a list of dimensions
    for j in range(m):
        for i in range(n):
            # append cropped image
            imgs.append(image.crop((i*w, j*h, (i+1)*w, (j+1)*h)))
    return imgs
def getImages(imageDir):
    """
    given a directory of images, return a list of Images
    """
    files = os.listdir(imageDir)
    images = []
    for file in files:
        filePath = os.path.abspath(os.path.join(imageDir, file))
        try:
            # explicit load so we don't run into a resource crunch
            fp = open(filePath, "rb")
            im = Image.open(fp)
            images.append(im)
            # force loading image data from file
            im.load()
            # close the file
            fp.close()
        except:
            # skip
            print("Invalid image: %s" % (filePath,))
    return images
def getImageFilenames(imageDir):
    """
    given a directory of images, return a list of image filenames
    """
    files = os.listdir(imageDir)
    filenames = []
    for file in files:
        filePath = os.path.abspath(os.path.join(imageDir, file))
        try:
            imgType = imghdr.what(filePath)
            if imgType:
                filenames.append(filePath)
        except:
            # skip
            print("Invalid image: %s" % (filePath,))
    return filenames
def getBestMatchIndex(input_avg, avgs):
    """
    return index of the best image match based on average RGB value distance
    """
    # input image average
    avg = input_avg
    # get the closest RGB value to input, based on RGB distance
    index = 0
    min_index = 0
    min_dist = float("inf")
    for val in avgs:
        dist = ((val[0] - avg[0])*(val[0] - avg[0]) +
                (val[1] - avg[1])*(val[1] - avg[1]) +
                (val[2] - avg[2])*(val[2] - avg[2]))
        if dist < min_dist:
            min_dist = dist
            min_index = index
        index += 1
    return min_index
def createImageGrid(images, dims):
    """
    given a list of images and a grid size (m, n), create a grid of images
```

```
    """
    m, n = dims
    # sanity check
    assert m*n == len(images)
    # get the maximum height and width of the images
    # don't assume they're all equal
    width = max([img.size[0] for img in images])
    height = max([img.size[1] for img in images])
    # create the target image
    grid_img = Image.new('RGB', (n*width, m*height))
    # paste the tile images into the image grid
    for index in range(len(images)):
        row = int(index/n)
        col = index - n*row
        grid_img.paste(images[index], (col*width, row*height))
    return grid_img
def createPhotomosaic(target_image, input_images, grid_size, reuse_images=True):
    """
    creates photomosaic given target and input images
    """
    print('splitting input image...')
    # split the target image into tiles
    target_images = splitImage(target_image, grid_size)
    print('finding image matches...')
    # for each tile, pick one matching input image
    output_images = []
    # for user feedback
    count = 0
    batch_size = int(len(target_images)/10)
    # calculate the average of the input image
    avgs = []
    for img in input_images:
        avgs.append(getAverageRGB(img))
    for img in target_images:
        # compute the average RGB value of the image
        avg = getAverageRGB(img)
        # find the matching index of closest RGB value
        # from a list of average RGB values
        match_index = getBestMatchIndex(avg, avgs)
        output_images.append(input_images[match_index])
        # user feedback
        if count > 0 and batch_size > 10 and count % batch_size is 0:
            print('processed %d of %d...' %(count, len(target_images)))
        count += 1
        # remove the selected image from input if flag set
        if not reuse_images:
            input_images.remove(match)
        print('creating mosaic...')
        # create photomosaic image from tiles
        mosaic_image = createImageGrid(output_images, grid_size)
        # display the mosaic
        return mosaic_image
    # gather our code in a main() function
def main():
    # command line arguments are in sys.argv[1], sys.argv[2], ...
    # sys.argv[0] is the script name itself and can be ignored
    # parse arguments
    parser = argparse.ArgumentParser(description='Creates a photomosaic from
                                     input images')
    # add arguments
    parser.add_argument('--target-image', dest='target_image', required=True)
    parser.add_argument('--input-folder', dest='input_folder', required=True)
    parser.add_argument('--grid-size', nargs=2, dest='grid_size', required=True)
    parser.add_argument('--output-file', dest='outfile', required=False)
    args = parser.parse_args()
    ###### INPUTS ######
    # target image
    target_image = Image.open(args.target_image)
    # input images
    print('reading input folder...')
    input_images = getImages(args.input_folder)
    # check if any valid input images found
    if input_images == []:
        print('No input images found in %s. Exiting.' % (args.input_folder, ))
        exit()
    # shuffle list to get a more varied output?
    random.shuffle(input_images)
    # size of the grid
```

```
    grid_size = (int(args.grid_size[0]), int(args.grid_size[1]))
    # output
    output_filename = 'mosaic.png'
    if args.outfile:
        output_filename = args.outfile
    # reuse any image in input
    reuse_images = True
    # resize the input to fit the original image size?
    resize_input = True
    ##### END INPUTS #####
    print('starting photomosaic creation...')
    # if images can't be reused, ensure m*n <= num_of_images
    if not reuse_images:
        if grid_size[0]*grid_size[1] > len(input_images):
            print('grid size less than number of images')
            exit()
    # resizing input
    if resize_input:
        print('resizing images...')
        # for given grid size, compute the maximum width and height of tiles
        dims = (int(target_image.size[0]/grid_size[1]),
                int(target_image.size[1]/grid_size[0]))
        print("max tile dims: %s" % (dims,))
        # resize
        for img in input_images:
        img.thumbnail(dims)
    # create photomosaic
    mosaic_image = createPhotomosaic(target_image, input_images, grid_size,
                                     reuse_images)
    # write out mosaic
    mosaic_image.save(output_filename, 'PNG')
    print("saved output to %s" % (output_filename,))
    print('done.')
# standard boilerplate to call the main() function
# to begin the program
if __name__ == '__main__':
    main()
```

Running the Photomosaic Generator

Here is a sample run of the program:

```
$ python photomosaic.py --target-image test-data/cherai.jpg --input-folder
test-data/set6/ --grid-size 128 128
reading input folder...
starting photomosaic creation...
resizing images...
max tile dims: (23, 15)
splitting input image...
finding image matches...
processed 1638 of 16384 ...
processed 3276 of 16384 ...
processed 4914 of 16384 ...
creating mosaic...
saved output to mosaic.png
done.
```

Figure 7-3(a) shows the target image, and Figure 7-3(b) shows the photomosaic. You can see a close-up of the photomosaic in Figure 7-3(c).

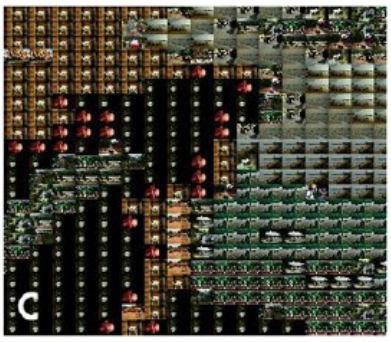

Figure 7-3: Photomosaic sample run

Summary

In this project, you learned how to create a photomosaic, given a target image and a collection of input images. When viewed from a distance, the photomosaic looks like the original image, but up close, you can see the individual images that make up the mosaic.

Experiments!

Here are some ways to further explore photomosaics.

1. Write a program that creates a blocky version of any image, similar to Figure 7-1.
2. With the code in this chapter, you created the photomosaic by pasting the matched images without any gaps in between. A more artistic presentation might include a uniform gap of a few pixels around each tile image. How would you create the gap? (Hint: factor in the gaps when computing the final image dimensions and when doing the paste in createImageGrid().)
3. Most of the time in the program is spent finding the best match for a tile from the input folder. To speed up the pro gram, getBestMatchIndex() needs to run faster. Your implementation of this method was a simple linear search through the list of averages (treated as three-dimensional points). This task falls under the general problem of a *nearest neighbor search*. One particularly effective way to find the closest point is a *k-d tree search*. The scipy library has a convenient class called scipy.spatial.KDTree, which lets you create a *k*-d and query it for the nearest point matches. Try replacing the linear search with a *k*-d tree using SciPy. (See *http://docs.scipy.org/doc/scipy/reference/generated/ scipy.spatial.KDTree.html*.)