```
from PIL import Image
import random
from sortedcontainers import SortedDict

def k_means(pixels, k):
    #pick random tuples between 0,width and 0,height for centers, use random.sample
    img_y = list(range(img_height))
    img_x = list(range(img_width))

    x_rand = random.sample(img_x, k)
    y_rand = random.sample(img_y, k)

    centers = []
    for i in range(k):
        #format: width, height, red, green, blue
        centers.append([pixels[x_rand[i], y_rand[i]][0], pixels[x_rand[i], y_rand[i]][1],
                pixels[x_rand[i], y_rand[i]][2]])

    done = False
    while(not done):
        cluster_list, mean_rgb_list = cluster_formation(centers, pixels, k)

        new_dists = []
        #tbd
        threshold = 5
        for j in range(k):
            dist = ((centers[j][0] - mean_rgb_list[j][0])**2
                + (centers[j][1] - mean_rgb_list[j][1])**2
                + (centers[j][2] - mean_rgb_list[j][2])**2)**(1/2)
            new_dists.append(dist)

        outside_threshold = False
        for x in new_dists:
            if(x > threshold):
                outside_threshold = True

        if(outside_threshold):
            centers = mean_rgb_list
        else:
            cluster_list, mean_rgb_list = cluster_formation(centers, pixels, k)
            done = True

    for d in range(k):
        for p in cluster_list[d].values():
            for s in range(len(p)):
                pixels[p[s][0], p[s][1]] = (mean_rgb_list[d][0], mean_rgb_list[d][1],
                mean_rgb_list[d][2])
```

```
    return pixels


def cluster_formation(centers, pixels, k):
    #iterate through pixels, finding which center is closest and creating clusters
    cluster_list = []
    for i in range(k):
        cluster_list.append(SortedDict())

    #iterating through all pixels in the image
    mean_rgb_list = [[0 for i in range(3)] for j in range(k)]

    for x in range(img_width):
        for y in range(img_height):
            min_dist = [1000000000, 0]
            #iterating through the 10 centers to find which is closest to each pixel
            for a in range(10):
                #calculating euclidean distance from point to center in terms of color values
                dist = ((centers[a][0] - pixels[x, y][0])**2
                        + (centers[a][1] - pixels[x, y][1])**2
                        + (centers[a][2] - pixels[x, y][2])**2)**(1/2)
                if(dist < min_dist[0]):
                    min_dist[0] = dist
                    min_dist[1] = a
            #populating clusters
            if(cluster_list[min_dist[1]].__contains__(min_dist[0])):
                cluster_list[min_dist[1]][min_dist[0]].append((x, y))
                mean_rgb_list[min_dist[1]][0] += pixels[x, y][0]
                mean_rgb_list[min_dist[1]][1] += pixels[x, y][1]
                mean_rgb_list[min_dist[1]][2] += pixels[x, y][2]
            else:
                cluster_list[min_dist[1]].setdefault(min_dist[0], [(x, y)])
                mean_rgb_list[min_dist[1]][0] += pixels[x, y][0]
                mean_rgb_list[min_dist[1]][1] += pixels[x, y][1]
                mean_rgb_list[min_dist[1]][2] += pixels[x, y][2]

    for t in range(k):
        total_vals = sum([len(v) for v in cluster_list[t].values()])
        mean_rgb_list[t][0] = round(mean_rgb_list[t][0] / total_vals)
        mean_rgb_list[t][1] = round(mean_rgb_list[t][1] / total_vals)
        mean_rgb_list[t][2] = round(mean_rgb_list[t][2] / total_vals)

    return cluster_list, mean_rgb_list

def SLIC(pixels):
    #Step 1: initialize a centroid in the center of 50x50 blocks
    center_x = 24
    center_y = 24
```

```
centroids = []
centroids.append([center_x, center_y, pixels[center_x, center_y][0],
            pixels[center_x, center_y][1], pixels[center_x, center_y][2]])
for x in range(24, img_width, 50):
    for y in range(24, img_height, 50):
        if(img_height - y < 50 and img_height-1-y != 25):
            center_y = center_y + (img_height - y) // 2
        elif(img_height-1-y == 25):
            break
        else:
            center_y = center_y + 50
        centroids.append([center_x, center_y, pixels[center_x, center_y][0],
                    pixels[center_x, center_y][1], pixels[center_x, center_y][2]])
    center_y = 24
    if(img_width - x < 50 and img_width-1-x != 25):
        center_x = center_x + (img_width - x) // 2
    elif(img_width-1-x == 25):
        break
    else:
        center_x = center_x + 50
    centroids.append([center_x, center_y, pixels[center_x, center_y][0],
                pixels[center_x, center_y][1], pixels[center_x, center_y][2]])

#Step 2: find RGB gradient in 3x3 around each center, move center to smallest one
sobel_centroids = RGB_gradient(centroids, pixels)

#Step 2: local shift
for c in range(len(sobel_centroids)):
    min = [10000, 0, 0]
    for r in sobel_centroids[c]:
        if(r[0] < min[0]):
            min = r
    centroids[c] = [min[1], min[2], pixels[min[1], min[2]][0],
                pixels[min[1], min[2]][1], pixels[min[1], min[2]][2]]

max_iter = 3
iterations = 0
converged = False
while(not converged and iterations < max_iter):
    #Step 3: assign each pixel to its nearest centroid in the 5d space of x/2, y/2, R, G, B

    #Dictionary with every point as a key with the value being the centroid it belongs to.
    #Key format is tuple (x, y) and value format is [x, y, dist]
    clusters_dict = {}
    #Dictionary with every centroid as a key with the values being every pixel in its
    #cluster. Key format is tuple (x, y) and value format is [[x1, y1, R1, G1, B1], ...]
    centroids_dict = {}
    for a in range(len(centroids)):
```

```
        x = centroids[a][0]-50
        y = centroids[a][1]-50
        centroids_dict[(centroids[a][0], centroids[a][1])] = []
        while(x < centroids[a][0]+51 and x < img_width):
          #boundary checking x
          if(x < 0):
            x = 0
          elif(x >= img_width):
            x = img_width-1

          while(y < centroids[a][1]+51 and y < img_height):
            #boundary checking y
            if(y < 0):
              y = 0

            #within 71 pixels
            within_71 = 71 > ((x - centroids[a][0])**2 + (y - centroids[a][1])**2)**(1/2)
            #finding euclidean distance
            if(within_71):
              dist_xy = (1/2) * (((x - centroids[a][0])**2) + ((y - centroids[a][1])**2))**(1/2)
              dist_rgb = ((pixels[x, y][0] - centroids[a][2])**2
                    + (pixels[x, y][1] - centroids[a][3])**2
                    + (pixels[x, y][2] - centroids[a][4])**2)**(1/2)
              dist = dist_xy + dist_rgb
              #checking if current pixel was already paired to a centroid
              if((x, y) in clusters_dict):
                if(clusters_dict[(x, y)][2] > dist):
                  #new centroid is closer than last, removing contributions to past
centroid's mean
                  centroids_dict[(clusters_dict[(x, y)][0], clusters_dict[(x, y)][1])].remove([x,
y, pixels[x, y][0], pixels[x, y][1], pixels[x, y][2]])
                  #adding point to closer centroid
                  clusters_dict[(x, y)] = [centroids[a][0], centroids[a][1], dist]
                  centroids_dict[(centroids[a][0], centroids[a][1])].append([x, y, pixels[x,
y][0], pixels[x, y][1], pixels[x, y][2]])
              else:
                clusters_dict[(x, y)] = [centroids[a][0], centroids[a][1], dist]
                centroids_dict[(centroids[a][0], centroids[a][1])].append([x, y, pixels[x, y][0],
pixels[x, y][1], pixels[x, y][2]])
            y += 1
          x += 1
          y = centroids[a][1]-50

      #Step 3: Computing new centroids
      comparator_dict = {}
      for t in centroids_dict.keys():
        new_centroid = [0, 0, 0, 0, 0]
        new_centroid[0] = round(sum(p[0] for p in centroids_dict[t]) / len(centroids_dict[t]))
```

```
            new_centroid[1] = round(sum(p[1] for p in centroids_dict[t]) / len(centroids_dict[t]))
            new_centroid[2] = round(sum(p[2] for p in centroids_dict[t]) / len(centroids_dict[t]))
            new_centroid[3] = round(sum(p[3] for p in centroids_dict[t]) / len(centroids_dict[t]))
            new_centroid[4] = round(sum(p[4] for p in centroids_dict[t]) / len(centroids_dict[t]))

            temp_x = t[0]
            temp_y = t[1]
            #format: key(x, y) : [[old_center], [new_center]]
            # [[x1, y1, R1, G1, B1], [x2, y2, R2, G2, B2]]
            comparator_dict[(temp_x, temp_y)] = [[temp_x, temp_y, pixels[temp_x, temp_y][0],
pixels[temp_x, temp_y][1], pixels[temp_x, temp_y][2]],
                                [new_centroid[0], new_centroid[1], new_centroid[2], new_centroid[3],
new_centroid[4]]]

        #Step 5: check for convergence
        difference_limit = 20
        converged = True
        for l in comparator_dict.keys():
            dist = (((comparator_dict[l][0][0] - comparator_dict[l][1][0])**2)/2
                    + ((comparator_dict[l][0][1] - comparator_dict[l][1][1])**2)/2
                    + (comparator_dict[l][0][2] - comparator_dict[l][1][2])**2
                    + (comparator_dict[l][0][3] - comparator_dict[l][1][3])**2
                    + (comparator_dict[l][0][4] - comparator_dict[l][1][4])**2)**(1/2)
            #print(dist)
            if(dist > difference_limit):
                converged = False
                break
        new_centroids = []
        for e in centroids:
            new_centroids.append(comparator_dict[(e[0], e[1])][1])

        centroids = new_centroids
        iterations += 1

    #Step 6: Coloring pixels
    for k in centroids_dict.keys():
        for p in centroids_dict[k]:
            pixels[(p[0], p[1])] = (comparator_dict[k][1][2], comparator_dict[k][1][3],
comparator_dict[k][1][4])

    #Step 6: coloring borders
    x = 0
    y = 0
    while(x < img_width):
        while(y < img_height):
            if(y < img_height-1 and pixels[x, y+1] != pixels[x, y] and pixels[x, y] != (0, 0, 0)):
                pixels[x, y] = (0, 0, 0)
                pixels[x, y+1] = (0, 0, 0)
```

```
                y += 2
            elif(x < img_width-1 and pixels[x+1, y] != pixels[x, y] and pixels[x, y] != (0, 0, 0)):
                pixels[x, y] = (0, 0, 0)
                pixels[x+1, y] = (0, 0, 0)
                y += 1
            else:
                y += 1
        y = 0
        if(x < img_width-1 and pixels[x+1, y-1] != pixels[x, y]):
            pixels[x, y] = (0, 0, 0)
            pixels[x+1, y] = (0, 0 ,0)
            x += 1
            y += 1
        else:
            x += 1
    return pixels


def RGB_gradient(centroids, pixels):
    sobel_x = [[1, 0, -1], [2, 0, -2], [1, 0, -1]]
    sobel_y = [[1, 2, 1], [0, 0, 0], [-1, -2, -1]]
    sobel_centroids = []
    sobel_sumx_red = 0
    sobel_sumy_red = 0
    sobel_sumx_green = 0
    sobel_sumy_green = 0
    sobel_sumx_blue = 0
    sobel_sumy_blue = 0

    for d in centroids:
        sobel_row = []
        #using sobel filters in a 3x3 around each point in a 3x3 around the centroid
        for e in range(-1, 2, 1):
            for f in range(-1, 2, 1):
                x = -1
                y = -1
                for g in range(3):
                    for h in range(3):
                        sobel_sumx_red += sobel_x[g][h] * pixels[d[0]+f+y, d[1]+e+x][0]
                        sobel_sumy_red += sobel_y[g][h] * pixels[d[0]+f+y, d[1]+e+x][0]
                        sobel_sumx_green += sobel_x[g][h] * pixels[d[0]+f+y, d[1]+e+x][1]
                        sobel_sumy_green += sobel_y[g][h] * pixels[d[0]+f+y, d[1]+e+x][1]
                        sobel_sumx_blue += sobel_x[g][h] * pixels[d[0]+f+y, d[1]+e+x][2]
                        sobel_sumy_blue += sobel_y[g][h] * pixels[d[0]+f+y, d[1]+e+x][2]
                        y += 1
                    x += 1
                    y = -1
                x = -1
```

```
            sobel_row_val = ((sobel_sumx_red)**2 + (sobel_sumy_red)**2
                        + (sobel_sumx_green)**2 + (sobel_sumy_green)**2
                        + (sobel_sumx_blue)**2 + (sobel_sumy_blue)**2)**(1/2)
            #format: gradient, width, height
            sobel_row.append([round(sobel_row_val), d[0]+f+y, d[1]+e+x])
#               if(d == [24, 24, 55, 75, 75]):
#                   print(sobel_row)
            sobel_sumx_red = 0
            sobel_sumy_red = 0
            sobel_sumx_green = 0
            sobel_sumy_green = 0
            sobel_sumx_blue = 0
            sobel_sumy_blue = 0
        sobel_centroids.append(sobel_row)
    return sobel_centroids

def main():
    img = Image.open("white-tower.png")

    #PIXELS IS IN FORMAT [WIDTH, HEIGHT]
    pixels_k = img.load()

    global img_height, img_width
    img_width = img.size[0]
    img_height = img.size[1]

    segmented_pixels = k_means(pixels_k, 10)
    img.save("k-means_tower.png")

    img2 = Image.open("wt_slic.png")

    pixels_slic = img2.load()

    img_width = img2.size[0]
    img_height = img2.size[1]

    slic_pixels = SLIC(pixels_slic)
    img2.save("SLIC_tower_borders.png")

if __name__ == "__main__":
    main()
```

This program takes 2 png images, "white-tower.png" and "wt_slic.png".
They must have those names or the program will not work. When running the program is does not require any inputs, as long as the images are in the same directory it will work properly.

The program will output 2 png images, "k-means_tower.png" and "SLIC_tower_borders.png",
however I have included a third image called "SLIC_tower.png" so it's easier to see the different clusters without all the noise from the black borders (moreso for the tower and tree areas).

The k-means implementation uses a list of 10 dictionaries (one for each center) with the keys being the distance from the centroid and the values for the keys being the points with that distance from the centroid. This implementation is a bit useless now as I originally needed the sorted dictionary to reassign centroids to the median point of the cluster, but the program now only uses the mean. K-means iterates through each pixel in the image and assigns them to the closest centroid with respect to RGB values, then it updates the centroids by taking the mean and checks to see if the new centroid is within a threshold of the old centroid. If it is, the program stops and if it isn't the process iterates again.

The SLIC implementation uses multiple dictionaries, one that has the centroids as keys and all the points, including their rgb values, in the cluster as values, and another that has every point as a key with its corresponding centroid and distance from the centroid as the single value. The method starts by evenly initializing centroids for 50x50 squares and then it takes the gradient around the centroids and locally shifts them. Next the proper loop begins and the program begins creating the clusters. Instead of iterating through each pixel, the loop iterates around the neighborhood of each centroid and assigns points. If a point has already been assigned to a centroid and a closer one is found, the dictionaries are updated to remove the point from the old centroid dictionary and to update the point's new centroid. After clusters are formed, new centroids are created from the mean x, y, and RGB of all the points in the cluster. The program checks to see if the new centroids are within a certain threshold of the old centroid just like the k-means method, but this method had a maximum of 3 iterations. Then the method colors the pixels of each cluster with the mean RGB value and colors pixels near neighboring clusters with black.