```python
from __future__ import division, print_function
import argparse
import matplotlib.image as mpimg
import matplotlib.pyplot as plt
import numpy as np
import os
import random


def init_centroids(num_clusters, image):
    """
    Initialize a `num_clusters` x image_shape[-1] nparray to RGB
    values of randomly chosen pixels of`image`

    Parameters
    ----------
    num_clusters : int
        Number of centroids/clusters
    image : nparray
        (H, W, C) image represented as an nparray

    Returns
    -------
    centroids_init : nparray
        Randomly initialized centroids
    """

    # *** START YOUR CODE ***
    # raise NotImplementedError('init_centroids function not implemented')
    H, W, C = np.shape(image)
    centroids_init = np.zeros(shape=[num_clusters, C])

    for idx in range(num_clusters):
        h, w = random.randint(1, H - 1), random.randint(1, W - 1)
        centroids_init[idx, :] = image[h, w, :]
    # *** END YOUR CODE ***

    return centroids_init


def update_centroids(centroids, image, max_iter=30, print_every=10):
    """
    Carry out k-means centroid update step `max_iter` times

    Parameters
    ----------
    centroids : nparray
        The centroids stored as an nparray
    image : nparray
        (H, W, C) image represented as an nparray
    max_iter : int
        Number of iterations to run
    print_every : int
        Frequency of status update

    Returns
    -------
    new_centroids : nparray
        Updated centroids
    """

    # *** START YOUR CODE ***
    # raise NotImplementedError('update_centroids function not implemented')
    H, W, C = np.shape(image)
    num_clusters = len(centroids)
    new_centroids = np.zeros(shape=[num_clusters, C])

    for it in range(max_iter):
        # Usually expected to converge long before `max_iter` iterations
        if it == 0 or (it + 1) % print_every == 0:
            print("[INFO] Completed iteration {} of {}".format(it + 1, max_iter))
        new_centroids = np.zeros(shape=[num_clusters, C])
        new_assignments = np.zeros(shape=[num_clusters, 1])

        for x in range(H):

            for y in range(W):
                # Initialize `dist` vector to keep track of distance to every centroid
                dist = np.zeros(shape=[num_clusters, 1])

                # Loop over all centroids and store distances in `dist`
                for idx in range(num_clusters):
                    d = centroids[idx, :] - image[x, y, :]
                    dist[idx] = np.dot(np.transpose(d), d)

                # Find closest centroid and update `new_centroids`
                centroid_idx = dist.argmin()
                new_assignments[centroid_idx] += 1
                new_centroids[centroid_idx, :] += image[x, y, :]

        # Update `new_centroids`
        for idx in range(num_clusters):
            if new_assignments[idx] > 0:
                new_centroids[idx, :] = new_centroids[idx, :] / new_assignments[idx]
    # *** END YOUR CODE ***

    return new_centroids


def update_image(image, centroids):
    """
    Update RGB values of pixels in `image` by finding
    the closest among the `centroids`

    Parameters
    ----------
    image : nparray
        (H, W, C) image represented as an nparray
    centroids : int
        The centroids stored as an nparray

    Returns
    -------
    image : nparray
        Updated image
    """

    # *** START YOUR CODE ***
    # raise NotImplementedError('update_image function not implemented')
    H, W, C = np.shape(image)
    num_clusters = len(centroids)

    for x in range(H):

        for y in range(W):
            # Initialize `dist` vector to keep track of distance to every centroid
            dist = np.zeros(shape=[num_clusters, 1])

            # Loop over all centroids and store distances in `dist`
            for idx in range(num_clusters):
                d = centroids[idx, :] - image[x, y, :]
                dist[idx] = np.dot(np.transpose(d), d)

            # Find closest centroid and update pixel value in `image`
            centroid_idx = dist.argmin()
            image[x, y, :] = centroids[centroid_idx]
    # *** END YOUR CODE ***

    return image


def main(args):

    # Setup
```

```python
    max_iter = args.max_iter
    print_every = args.print_every
    image_path_small = args.small_path
    image_path_large = args.large_path
    num_clusters = args.num_clusters
    figure_idx = 0

    # Load small image
    image = np.copy(mpimg.imread(image_path_small))
    print('[INFO] Loaded small image with shape: {}'.format(np.shape(image)))
    plt.figure(figure_idx)
    figure_idx += 1
    plt.imshow(image)
    plt.title('Original small image')
    plt.axis('off')
    savepath = os.path.join('.', 'orig_small.png')
    plt.savefig(savepath, transparent=True, format='png', bbox_inches='tight')

    # Initialize centroids
    print('[INFO] Centroids initialized')
    centroids_init = init_centroids(num_clusters, image)

    # Update centroids
    print(25 * '=')
    print('Updating centroids ...')
    print(25 * '=')
    centroids = update_centroids(centroids_init, image, max_iter, print_every)

    # Load large image
    image = np.copy(mpimg.imread(image_path_large))
    image.setflags(write=1)
    print('[INFO] Loaded large image with shape: {}'.format(np.shape(image)))
    plt.figure(figure_idx)
    figure_idx += 1
    plt.imshow(image)
    plt.title('Original large image')
    plt.axis('off')
    savepath = os.path.join('.', 'orig_large.png')
    plt.savefig(fname=savepath, transparent=True, format='png', bbox_inches='tight')

    # Update large image with centroids calculated on small image
    print(25 * '=')
    print('Updating large image ...')
    print(25 * '=')
    image_clustered = update_image(image, centroids)

    plt.figure(figure_idx)
    figure_idx += 1
    plt.imshow(image_clustered)
    plt.title('Updated large image')
    plt.axis('off')
    savepath = os.path.join('.', 'updated_large.png')
    plt.savefig(fname=savepath, transparent=True, format='png', bbox_inches='tight')

    print('\nCOMPLETE')
    plt.show()


if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('--small_path', default='./peppers-small.tiff',
                        help='Path to small image')
    parser.add_argument('--large_path', default='./peppers-large.tiff',
                        help='Path to large image')
    parser.add_argument('--max_iter', type=int, default=150,
                        help='Maximum number of iterations')
    parser.add_argument('--num_clusters', type=int, default=16,
                        help='Number of centroids/clusters')
    parser.add_argument('--print_every', type=int, default=10,
                        help='Iteration print frequency')
    args = parser.parse_args()
    main(args)
```