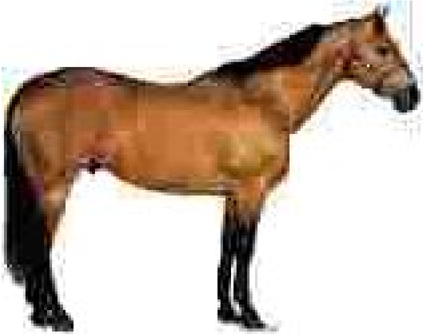
Contents

1 Basic Test Results

```
Starting tests...
Wed Dec  2 20:10:35 IST 2015
87fc38901f08b853b1303110026638fef66e1bd1  -


Archive:  /tmp/bodek.rfUboC/intro2cs/ex6/elinorperl/presubmission/submission
  inflating: src/ex6.py
  inflating: src/README

Testing README...
Done testing README...

Running presubmit tests...
19 passed tests out of 19
result_code    ex6    19    1
Done running presubmit tests

Showing execution with wrong number of parameters:
Wrong number of parameters. The correct usage isex6.py <image_source> <images_dir><output_name> <tile_height><num_candidates
Done...

Showing correct execution:
Done...

Tests completed

Additional notes:

There will be additional tests which will not be published in advance.
```

2 horse.jpg



3 README

```
elinorperl
329577464
Elinor Perl

I discussed the exercise with Talya Adams, Yasmin Yusobov, and lab support

=========================================
=  README for ex6: Making Mosaics  =
=========================================


==================
=  Description:  =
==================

In this exercise, I created a program that takes an image, list of tiles and size input,
and creates a mosaic picture. The program uses the average color of the pictures, and compares
their distances in order to see what picture is most suited for the spot. According to the
averages and distance, it takes the best values and creates a list of the best options to use
for the mosaic, thereafter choosing the best one. At this point the mosaic can be created!

======================
=  Special Comments  =
======================

I used stackoverflow.com
```

4 ex6.py

```python
import mosaic
import sys
import copy

NUMBER_OF_ARGUMENTS = 5
ERROR_MESSAGE = 'Wrong number of parameters. The correct usage is\
ex6.py <image_source> <images_dir><output_name> <tile_height>\
<num_candidates>'


def compare_pixel(pixel1, pixel2):
    """Compares the distance of two pixels by calculating the sum of the absolute
    value of the difference of each color in the rgb.
    """
    red1, green1, blue1 = pixel1
    red2, green2, blue2 = pixel2
    pixel_distance = abs(red1 - red2) + abs(green1 - green2) + abs(blue1 - \
                                                               blue2)
    return pixel_distance


def compare(image1, image2):
    """ Compares the average of two pictures on the same  principle of the
    for function (compare_pixel).
    """
    image_distance = 0
    for row in range(min(len(image1), len(image2))):
        for column in range(min(len(image1[0]), len(image2[0]))):
            image_distance += compare_pixel(image1[row][column], image2[row]\
                [column])
    return image_distance


def get_piece(image, upper_left, size):
    """
    Returns a cropped picture according to the components defined. The cropped
    picture starts as an empty list adding on from the upper left point
    throughout the size that was given, creating a cropped picture, ending
    either at the end of the size or the end of the picture (whichever ends
    first).
    """
    cropped_image = []
    point1, point2 = upper_left
    height = size[0]
    width = size[1]
    endpoint1 = point1 + height
    endpoint2 = point2 + width
    min_endpoint1 = min(endpoint1,len(image))
    min_endpoint2 = min(endpoint2, len(image[0]))
    for row in range(point1, min_endpoint1):
        cropped_image.append(image[row][point2:min_endpoint2])
    return cropped_image


def set_piece(image, upper_left, piece):
    """ Placing the "piece" starting from the upper left point throughout it's
    size on the picture. I defined the minimum end of the loops (either the
    picture or the piece). Using the nested loop throughout the upper left to
    the minimum point I defined, morphing the image into the piece.
    """
    starting_row, starting_column = upper_left
    piece_height = len(piece)
    piece_width = len(piece[0])
    endpoint_height = min(len(image),starting_row + piece_height)
    endpoint_width = min(len(image[0]),starting_column + piece_width)
    for i in range(starting_row,endpoint_height):
        for j in range(starting_column,endpoint_width):
            image[i][j] = piece[i-starting_row][j-starting_column]


def average(image):
    """
    Takes a the rgb of the whole picture and calculates the average of each
    color, by adding the reds, greens, and blues of the whole image and
    dividing them by the amount of rows and columns (translating into the
    amount of tuples containing the rgb in the image), returning the each
    average in a tuple.
    """
    red_average = 0
    green_average = 0
    blue_average = 0
    for i in range(len(image)):
        for j in range(len(image[0])):
            red_average += image[i][j][0]
            green_average += image[i][j][1]
            blue_average += image[i][j][2]
    red_average = float(red_average/(len(image)*len(image[0])))
    green_average = float(green_average/(len(image)*len(image[0])))
    blue_average = float(blue_average/((len(image))*len(image[0])))
    return (red_average, green_average, blue_average)


def preprocess_tiles(tiles):
    """
    Creates a list of averages for each tile.
    """
    average_list = []
    for tile in tiles:
        average_list.append(average(tile))
    return average_list


def new_min(tiles):
    """
    A helpful function for get_best_tiles, creating a new minimum each time.
    """
    next_best_i = 0
    for i in range(1,len(tiles)):
        if tiles[i] < tiles[next_best_i]:
            next_best_i = i
    return next_best_i


def get_best_tiles(objective, tiles, averages, num_candidates):
    """
    I created a list comparing the objective average and averages given.
    Until the minimum value list reaches the length of the num_candidates, it
    checks the new minimum from the function I made before, and adds in to my
    list of minimum values, deleting it from the other lists.
    """
    min_values = []
    distance_list = []
    tile_copies = tiles[:]
    objective_average = average(objective)
    for i in range(len(tiles)):
        i_distance = compare_pixel(objective_average,averages[i])
        distance_list.append(i_distance)
    while len(min_values) < num_candidates:
        min_index = new_min(distance_list)
        min_values.append(tile_copies[min_index])
        del tile_copies[min_index]
        del distance_list[min_index]
    return min_values


def choose_tile(piece, tiles):
    """
    Choose tile chooses the smallest value from the tiles by comparing each
    tile with the piece and if the comparison is smaller than the previous it
    will return the smallest tile.
    """
    min_tile = tiles[0]
    min_comparison = compare(tiles[0],piece)
    for tile in tiles:
        comparison = compare(tile, piece)
        if comparison < min_comparison:
            min_tile = tile
            min_comparison = comparison
    return min_tile


def make_mosaic(image, tiles, num_candidates):
    """
    In a nested loop throughout the picture and jumping one tile at a time,
    the loop calls the former functions, to make the mosaic, using the
    coefficients we input.
    """
    copied_image = copy.deepcopy(image)
    tile_average = preprocess_tiles(tiles)
    tile_height = len(tiles[0])
    tile_width = len(tiles[0][0])
    for row in range(0, len(copied_image), tile_height):
        for column in range(0, len(copied_image[0]), tile_width):
            piece = get_piece(copied_image, (row, column),\
                              (tile_height, tile_width))
            best_tiles = get_best_tiles(piece, tiles, tile_average,\
                                        num_candidates)
            set_piece(copied_image, (row, column), choose_tile(piece, \
                                                               best_tiles))
    return copied_image


if __name__ == "__main__":
    """
    The main calls the outer elements (images, tiles, etc) into the program if
    the sys.argv doesn't exceed the number of arguments, and in the case it
     does it will print an error message.
    """
    if len(sys.argv) != NUMBER_OF_ARGUMENTS + 1:
        print(ERROR_MESSAGE)
    else:
        script_name = sys.argv[0]
        image_source = sys.argv[1]
        images_dir = sys.argv[2]
        output_name = sys.argv[3]
        tile_height = int(sys.argv[4])
        num_candidate = int(sys.argv[5])
        image = mosaic.load_image(image_source)
        tiles = mosaic.build_tile_base(images_dir, tile_height)
        mosaic.save(make_mosaic(image, tiles, num_candidate), output_name)
```