Next, I decided that analyzing a full rectangle would be inefficient, because it would probably contain unnecessary data around the corners. In order to get away from this issue I calculated how to use 8 rectangles of various sizes, so that I can cover a good portion of the initial rectangle, which is at its biggest 1200 x 1600.

Analyzing the photo is done in stages. Since the image downloaded will always be with a resolution of 3264 x 4928 pixels, it is easy to establish the center points of the five rectangles.

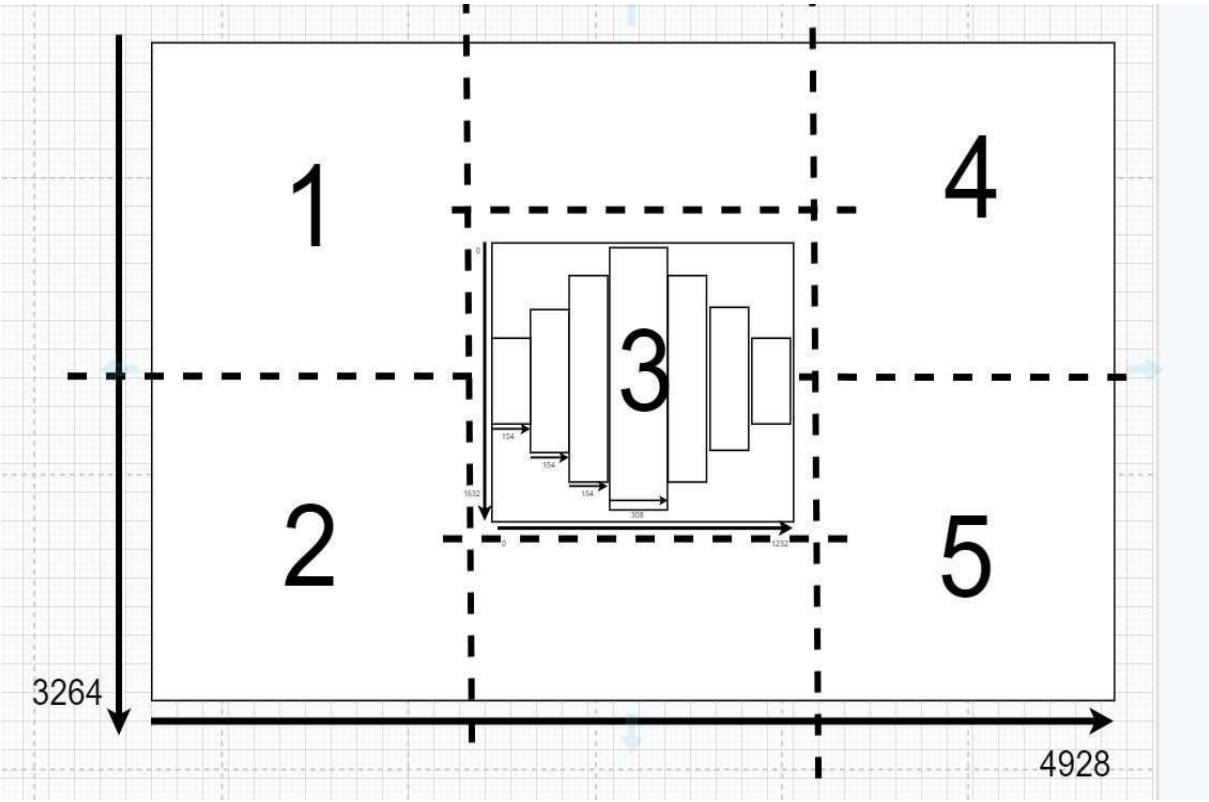

Figure 23. Position of the five "circles." The total photo is split into 5 boxes, with one in the center, which will have the most priority.

The reason why I wanted to begin with 5 predetermined spots and rectangles is because if I had to use random positions, my data could be overlapping in some spaces and that is just more work to make sure that any overlapping scenarios will be distinguished. Working in separate environments, would also provide a good amount of surface area.

**There are small margins on the left and right side of the photo.

Box 1 center:	1232 x 816
Box 2 center:	1232 x 2448
Box 3 center:	2464 x 1632
Box 4 center:	3696 x 816
Box 5 center:	3696 x 2448

Based on the center point of each rectangle, a function is called to calculate the positions of each rectangle. Inside the Circle class, one of the methods is called to do the calculation for each rectangle.

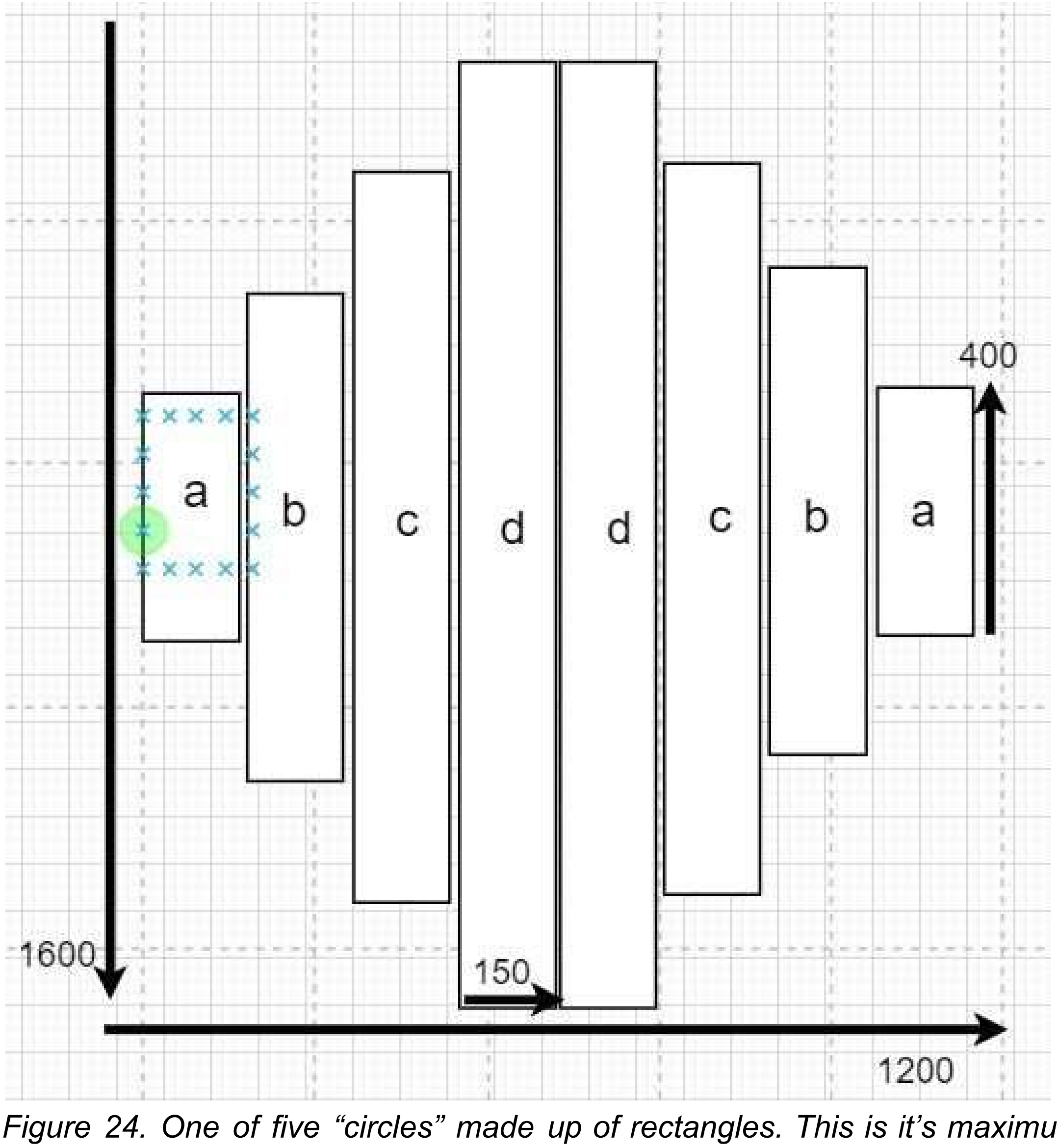

Figure 24. One of five "circles" made up of rectangles. This is it's maximum size in pixels

From left to right, the size of the rectangles are 400x150, 800x150, 1200x150, 1600x150. The right side would be the same. A variable current_scale = # is used to change the size of the circles cropped from the photo. This is why the maximum size is 1200 x 1600. The current_scale variable ranges from 1 to 10, which in terms is applied to the method make_circle(img, circle_position, side, scale). The img variable is the name of the photo, circle_position is concerned with which rectangle center will be used (Figure 5). Since the circle is made up of 8 rectangles, 2 identical but mirrored halves, the side variable represents which side is going to be invoked. Finally, the scale determines the overall size of the circles.

I have calculated that with this method, at its smallest size, one circle covers 6.2% of each rectangle and 3.7% of the total photo. At its biggest size, the circle covers 62.5% of each rectangle and 37.3% of the overall photo. Each circle, at maximum size, contains 1 200 000 pixels and the two rectangles in the middle of each circle cover the first 25%. By my assumptions, this percentage should be enough to make a good guess as to what are the dominating colors in the picture. There is the potential risk of having unnecessary data, so this is why I have made these calculations.

separate_circles.py

```
from multiprocessing import Process,Queue,Pipe
from PIL import Image
import numpy as np
from collections import Counter

class Circle:
   global centers_list
   global most_common_red
   centers_list = [(1232, 816), (1232, 2448), (2464, 1632), (3696, 816), (3696, 2448)]
   global reds
   global greens
   global blues
   reds = list()
   red = []
   greens = list()
   greens = []
   blues = list()
   blues = []


   def make_circle(img, circle_position, side, scale):
      img_open = Image.open(img)
      current_scale = scale
      current_side = side
      x = circle_position - 1
      n = 4
      n_opposite = 1
      for x_two in range(0,4):
         if current_side == 0:
         #distinguish coordinates for four rectangles
            rect_one_x_one = centers_list[x][0] - (n * (15 * scale))
            rect_one_y_one = centers_list[x][1]- (n_opposite * (20 * scale))
            rect_one_x_two = centers_list[x][0] - ((n-1) * (15 * scale))
            rect_one_y_two = centers_list[x][1] + ((n_opposite) * (20 * scale))
            elif current_side == 1:
                rect_one_x_one = centers_list[x][0] + ((n-1) * (15 * scale))
                rect_one_y_one = centers_list[x][1]- (n_opposite * (20 * scale))
                rect_one_x_two = centers_list[x][0] + ((n) * (15 * scale))
         rect_one_y_two = centers_list[x][1] + ((n_opposite) * (20 * scale))

#call cropping def
 crop_box = (rect_one_x_one, rect_one_y_one, rect_one_x_two, rect_one_y_two)
            region_temp = img_open.crop(crop_box)
            data_segment = np.array(region_temp.getdata())

            end = int(len(data_segment))
            #populates the list with B colors so they can be analyzed
            #min, max, occurrences
            for i_temp in range(0, end):
                temp = data_segment[i_temp]
                blues.append(temp[2])
                reds.append(temp[0])
                greens.append(temp[1])

            n -= 1
            n_opposite += 1

            if n == 0:
                n = 4
            if n_opposite == 5:
                n_opposite = 1


    def show_trends():
        cnt_blue = Counter(blues)
        cnt_red = Counter(reds)
        cnt_green = Counter(greens)

        reds[:] = []
        greens[:] = []
        blues[:] = []

        most_common_red = str(cnt_red.most_common(5)[0][0])
        m_c_red = int(most_common_red)
        most_common_green = str(cnt_green.most_common(5)[0][0])
        m_c_green = int(most_common_green)
        most_common_blue = str(cnt_blue.most_common(5)[0][0])
        m_c_blue = int(most_common_blue)

        return (m_c_red, m_c_green, m_c_blue)
```

The Circle class begins with establishing variables such as global lists for most common values for each color and the centers of each of the five segments of the photo. Each list is initiated with _list_ = [], because if it is not established, I am unable to use it later and I do need those values to be global since they are used in other methods. Inside the make_circle() method, the name of the JPG file, the side, position and scale are given. More specific variables for this method are the n, n_opposite, and x. X simply fetches the coordinates of the center, so that the function for the circles would know where to start from. N starts from 4 and at the end of each for loop it goes down by 1, after reaching 1, it goes back to 4. This means that the function draws the leftmost rectangle first (smallest one), and the middle one (largest one) last. As the name suggests n_opposite, moves in the opposite way of n. N plays a role in the width of the rectangle and n_opposite plays a role in the height.

Depending on which side was chosen (left 0, right 1), different mathematical functions are used, which are inside a for loop that is repeating 4 times Examining the left side function for the middle rectangle:

```
rect_one_x_one = centers_list[x][0] - (n * (15 * scale))
rect_one_y_one = centers_list[x][1]- (n_opposite * (20 * scale))
rect_one_x_two = centers_list[x][0] - ((n-1) * (15 * scale))
rect_one_y_two = centers_list[x][1] + ((n_opposite) * (20 * scale))
```

Showing manually

```
Loop 1:
Rect_one_x_one = 2464 - (4 * (15 * 1))
Rect_one_y_one = 1632 - (1 * (20 * 1))
Rect_one_x_two = 2464 - (3 * (15 * 1))
Rect_one_y_two = 1632 + (1 * (20 * 1))

Rect_one_x_one = 2404
        Rect_one_y_one = 1612
                Rect_one_x_two = 2419
                        Rect_one_y_one = 1652

Loop 2:
        Rect_one_x_one = 2419
                Rect_one_y_one = 1592
                        Rect_one_x_two = 2434
                                Rect_one_y_one = 1672

Loop 3:
        Rect_one_x_one = 2430
                Rect_one_y_one = 1572
                        Rect_one_x_two = 2449
                                Rect_one_y_one = 1692
Loop 4
        Rect_one_x_one = 2449
                Rect_one_y_one = 1552
                        Rect_one_x_two = 2464
```

Rect_one_y_one = 2112

After each loop, the coordinates are passed to the crop() method and the data is transferred to an array using np.array(region_temp.getdata()). Pillow has a method __.show() which provides a representation of the cropped segments and what the computer sees.

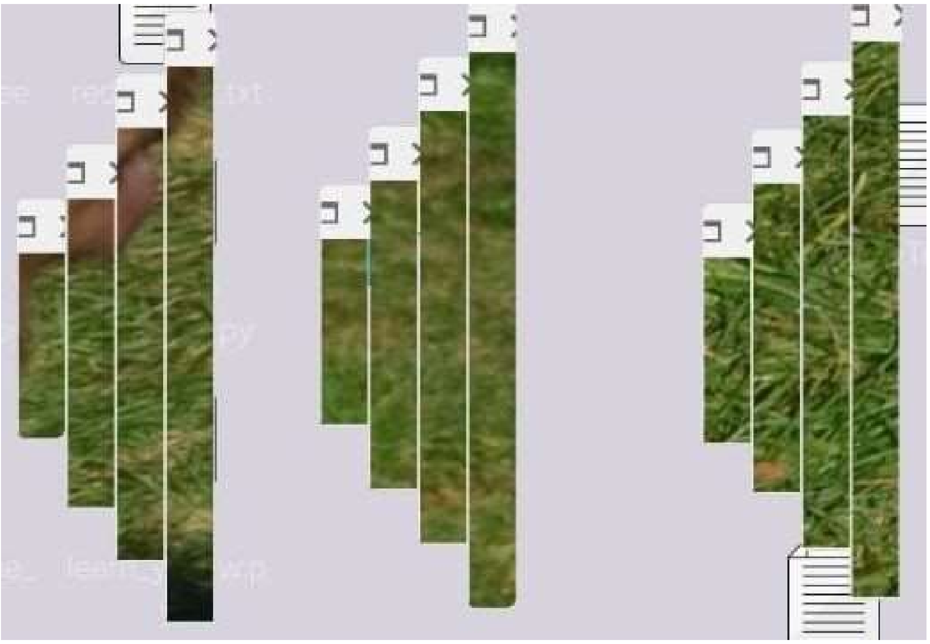

Figure 25 shows 3 left sided halves taken from one photo and how they are represented with the .show() method

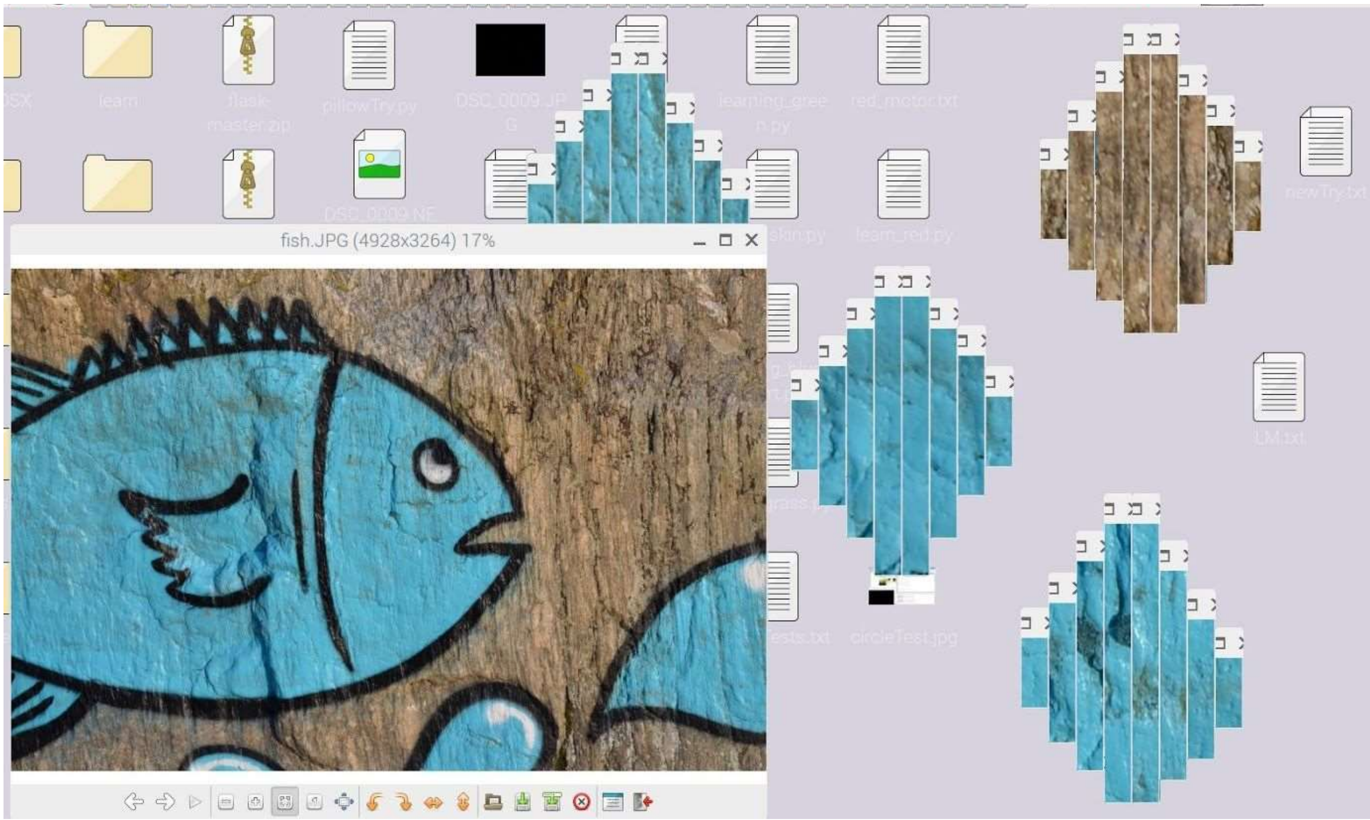

Figure26 *A screenshot from Raspbian, showing four complete circles and the original photo that they were cropped from. Bottom left circle (fifth one) is covered so I could fit the picture in the screenshot.*

Once the segments have been cropped a for loop is invoked for the length of the data_segment. The way the information is stored in the list is on index 0, the value is [0, 0 , 0] ([R, G, B]). Inside this for loop each color list is populated with its values

```
 for i_temp in range(0, end):
        temp = data_segment[i_temp]
        blues.append(temp[2])
        reds.append(temp[0])
        greens.append(temp[1])
```

The previously mentioned make circle method is called inside the Scanner class and inside a for loop that is run 5 times. Circle.make_circle() is called two times for the two respective sides, left and right.

Analyze_sides.py from initial_scan()

```
circle = Circle
scan = Scanner
temp_location = location_pic + path
for i in range (0,5):
     circle.make_circle(temp_location, i, 1, scale_default)
```