Manipulating Colors

In the previous section we saw how to change the red, green, and blue values of one pixel. We then learned that we could use a loop to change the color of all the pixels in one row, and we could use nested loops to change the color of all of the pixels in an image. We are going to explore that more now.

Recall the following function from the previous reading that turns all pixels black:

Example 1: Color all pixels using nested loops

```
def colorAllBlack(picture):
  # Duplicate the original picture
  newPic = picture.copy()

  for y in range(newPic.height):
    # Change the pixels in row y
    for x in range(newPic.width):
      newPic.putpixel((x,y),(0, 0, 0))

  # Return the new picture
  return newPic
```

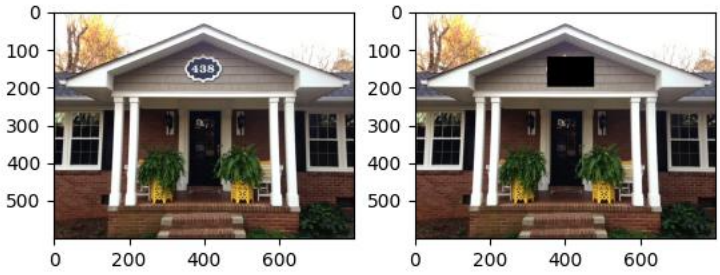
Turning all of the pixels black may seem a little drastic; changing the ranges used in the loops could allow us to color only a segment of the image black, such as what might be used to cover a license plate, or a house address, or someone's face when we should not be making that information public.

The following function changes just a portion of the picture to black, to cover the address at the top of the house. The house image was first shown as a plotted figure, so that we could determine the coordinates of the area to black out.
The coordinate of the top left corner of a rectangle looks to be (350, 120), and the bottom right corner looks to be (475, 200).

Example 2: Color only a selection of pixels using nested loops

```
def colorSomeBlack(picture):
  # Duplicate the original picture
  newPic = picture.copy()

  for y in range(startY, endY):
    # Change the pixels in row y
    for x in range(startX, endX):
      newPic.putpixel((x,y),(0, 0, 0))

  # Return the new picture
  return newPic
```

Below are the results of running this function on the house picture, with the coordinates (`startX, startY`) = (350, 120), and (`endX, endY`) = (475, 200). The original picture is on the left, the modified picture is on the right.



Instead of changing the values of red, green, and blue to 0 for each pixel, we may like to change only red, or only green, or only blue. Or we may like to change each of red, green, and blue, but by different amounts.

The following examples show how different effects may be achieved. You should look at how these functions work – how are they getting pixels and changing the color values? Then try these in your own program.

The following example is a function to cut the red value of a picture by a quarter.

Example 3: Reducing red by 25%

```
# Function to reduce red values by 25%
def reduceRed(picture):
  newPic = picture.copy()
  for x in range(newPic.width):
    for y in range(newPic.height):
          rvalue, gvalue, bvalue = newPic.getpixel((x, y))
          newR = int(rvalue * 0.75)
          newPic.putpixel((x, y),(newR, gvalue, bvalue))

  return newPic
```

Notice that the loop structure in the middle of this function looks exactly like the loop we used in the previous example. The code inside the loop is slightly different - here we get the current amount of red in a pixel and then change it. By multiplying the red value by a factor less than 1 we are making the red value smaller. Reducing the color by 25% means we have 75% of the color remaining. That's why we multiply the current red value by 0.75. The values for red, green, and blue must be integers, so we *cast* the new red value by passing the result of the calculation to the `int` function. This will convert a decimal value into an integer by ignoring anything after the decimal point.

As in the example where we changed all the pixels to black, we see in this example that a picture gets passed in as a parameter, and then the first thing we do is duplicate it. We do this so that we don't modify the original picture. We create a new picture identical to the original, and then make our changes to the new picture. At the end of the function, we return the new picture so that we can use or save our results.

In a Code cell, we would call this function with code something like the following:

```
myImage = Image.open('/drive/MyDrive/ColabNotebooks/arch.jpg')
newImage = reduceRed(myImage)
newImage.show()
```

How could we modify this function so that we make the red values larger? The only change we need to make is to change the multiplication factor to be a value larger than 1 instead of smaller than 1.

Example 4: Increase the amount of red in a picture

```
# Function to increase red values by 20%
def increaseRed(picture):
  newPic = picture.copy()
  for x in range(newPic.width):
    for y in range(newPic.height):
          rvalue, gvalue, bvalue = newPic.getpixel((x,y))
          newR = int(rvalue * 1.2)
          newPic.putpixel((x,y),(newR, gvalue, bvalue))

  return newPic
```

What happens if you increase the red in a picture that has a lot of red? When you multiply the red values by something larger than 1, there is a chance that the resulting value will be greater than 255. So what should we do? One possibility is that the program could crash. Another option would be to clip (*i.e.*, cap) the value of red at 255, and still another option would be to wrap it around using the modulo (remainder) operator. Wrapping around may give you unexpected (but possibly interesting) results. For example, if the red value was 150 and you tried to double the red, the new value would be 300. We can't have a red value of 300, so we would either set it to 255, or let it wrap around to 44 (300 – 256). The `putpixel` function of an `Image` object in the PIL library is designed to cap the values at 255. If you attempt to set the color of a pixel with values greater than 255, the value 255 will be used.

So far, we have only done one color manipulation in the loop. There is nothing preventing us from doing more than one color manipulation at time. Suppose we want to add a sunset effect to our picture. When you view a sunset, the sky seems to redden, while everything gets darker. One way to do this could be to reduce the amount of green and blue in the picture. By doing this, the red will stand out more, and the other colors will get darker.

Example 5: Sunset Effect

```
# make sunset
def sunset(picture):
  # duplicate the original picture
  newPic = picture.copy()

  # reduce the green and blue of all pixels
  for y in range(newPic.height):
    for x in range(newPic.width):
      rvalue, gvalue, bvalue = newPic.getpixel((x,y))
      newg = int(gvalue * 0.7)
      newb = int(bvalue * 0.7)
      newPic.putpixel((x,y),(rvalue, newg, newb))

  # return the new picture
  return newPic
```

Notice, again, in this example, we are using the same loop structure to iterate through and change each of the pixels in the pictures. The big difference here is that we are getting and changing both the green and the blue values inside the loop, while keeping the red the same.

Creating Negatives

To create the negative of an image, we want the opposite of each the current values of red, green, and blue. So what does this mean? If we have no red (*i.e.*, the red value is 0), we need all red (*i.e.*, a red value of 255). If there is a lot of red, we need a little bit of red, and vice versa. So, say the red value of a pixel in a picture is 60. The red value in the corresponding pixel in the negative image would be 255 – 60 = 195. So, to create a picture that is the negative, we compute the negative value (255 – original value) of each of the red, green, and blue components for each pixel in the original picture. We then set the colors of the pixels in the new picture to these new colors. Our function would look like:

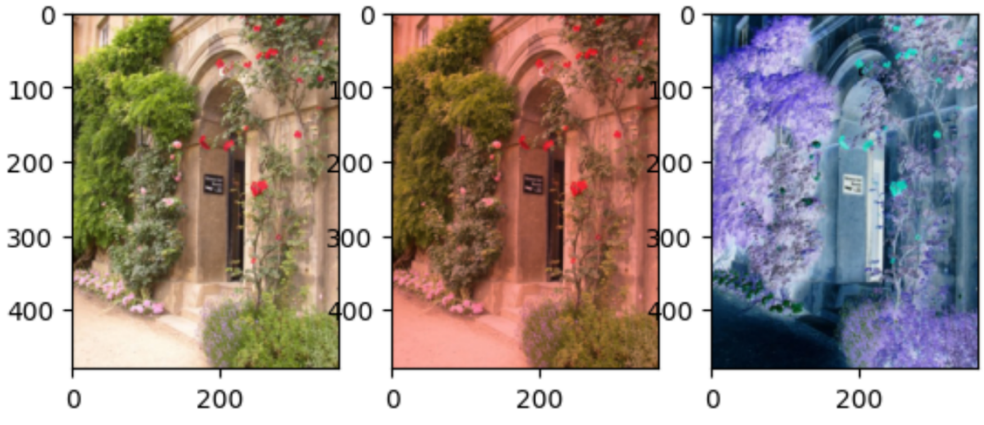
Example 6: Creating a negative image

```
# make a sunset effect
def negative(picture):
  # duplicate the original picture
  newPic = picture.copy()

  # reduce the green and blue of all pixels
  for y in range(newPic.height):
    for x in range(newPic.width):
      rvalue, gvalue, bvalue = newPic.getpixel((x,y))
      newPic.putpixel((x,y),(255-rvalue, 255-gvalue, 255-bvalue))

  # return the new picture
  return newPic

   return newPict
```

Calling the sunset effect and then the negative effect on an image and then displaying the results would look something like the following:

```
1 newImage = sunset(myImage)
2 newImage2 = negative(myImage)
3 fig, pics = plt.subplots(1,3)
4 pics[0].imshow(myImage)
5 pics[1].imshow(newImage)
6 pics[2].imshow(newImage2)
7
```

```
<matplotlib.image.AxesImage at 0x7fb2a8ebd640>
```



Creating Grayscale Images

Creating grayscale images is also not too difficult. When the red component, green component, and blue component all have the same value, the resultant color is gray. This means that we will have 256 different shades of gray, ranging from black at (0, 0, 0), to white at (255, 255, 255). The only tricky part is figuring out what the replicated value should be. We want a sense of the intensity of the color, or the luminance. One way to compute this is to compute the average of the red, green, and blue component colors. Our function would look like the following:

Example 6: Creating a grayscale image

```
# make grayscale effect
def grayscale(picture):
  # duplicate the original picture
  newPic = picture.copy()

  # reduce the green and blue of all pixels
  for y in range(newPic.height):
    for x in range(newPic.width):
      rvalue, gvalue, bvalue = newPic.getpixel((x,y))
      intensity = (rvalue + gvalue + bvalue)//3
      newPic.putpixel((x,y),(intensity, intensity, intensity))

  # return the new picture
  return newPic
```

As it turns out, this method really oversimplifies the notion of grayscale. We can actually take into account how the human eye perceives luminance – we consider blue to be darker than red, even if there is the same amount of light reflected off. So we will weight blue lower, red and green higher when we compute the average. Our modified function would look like:

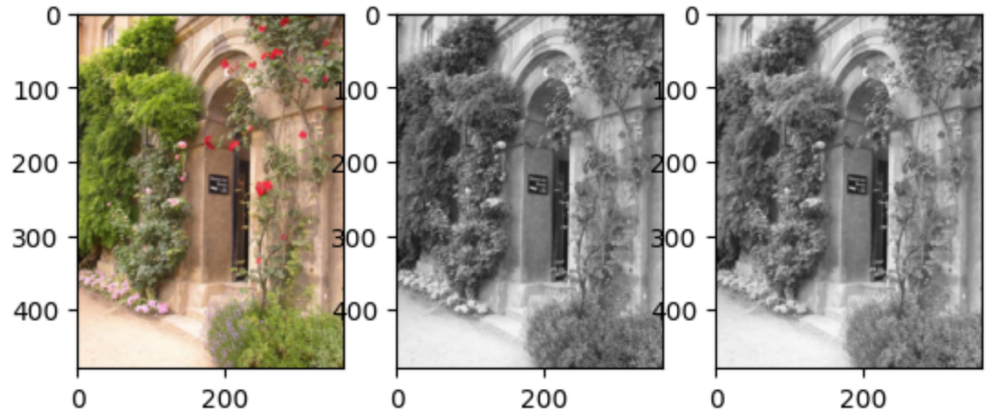
Example 7: Grayscale with weights

```
# make weighted grayscale effect
def weightedgrayscale(picture):
  # duplicate the original picture
  newPic = picture.copy()

  # reduce the green and blue of all pixels
  for y in range(newPic.height):
    for x in range(newPic.width):
      rvalue, gvalue, bvalue = newPic.getpixel((x,y))
      newr = rvalue * 0.299
      newg = gvalue * 0.587
      newb = bvalue * 0.114
      luminance = int(newr + newg + newb)
      newPic.putpixel((x,y),(luminance, luminance, luminance))

  # return the new picture
  return newPic
```

The results of calling these functions are below (grayscale is in the middle, weighted grayscale is on the right):

```
1 newImage = grayscale(myImage)
2 newImage2 = weightedgrayscale(myImage)
3 fig, pics = plt.subplots(1, 3)
4 pics[0].imshow(myImage)
5 pics[1].imshow(newImage)
6 pics[2].imshow(newImage2)
```

```
<matplotlib.image.AxesImage at 0x7fb2a8fb88e0>
```



You may, of course, experiment with the weights to obtain different results.

We will now test some of these functions and write some of our own functions to manipulate pixel colors in the next activity.

Activity: Manipulating Colors