

Python Code Documentation for the Project

This project was developed in the Python Language, using the Python Image Library (PIL) for photo modification and the AppJar Graphical User Interface (GUI) module to host all of the methods involved in making this application.

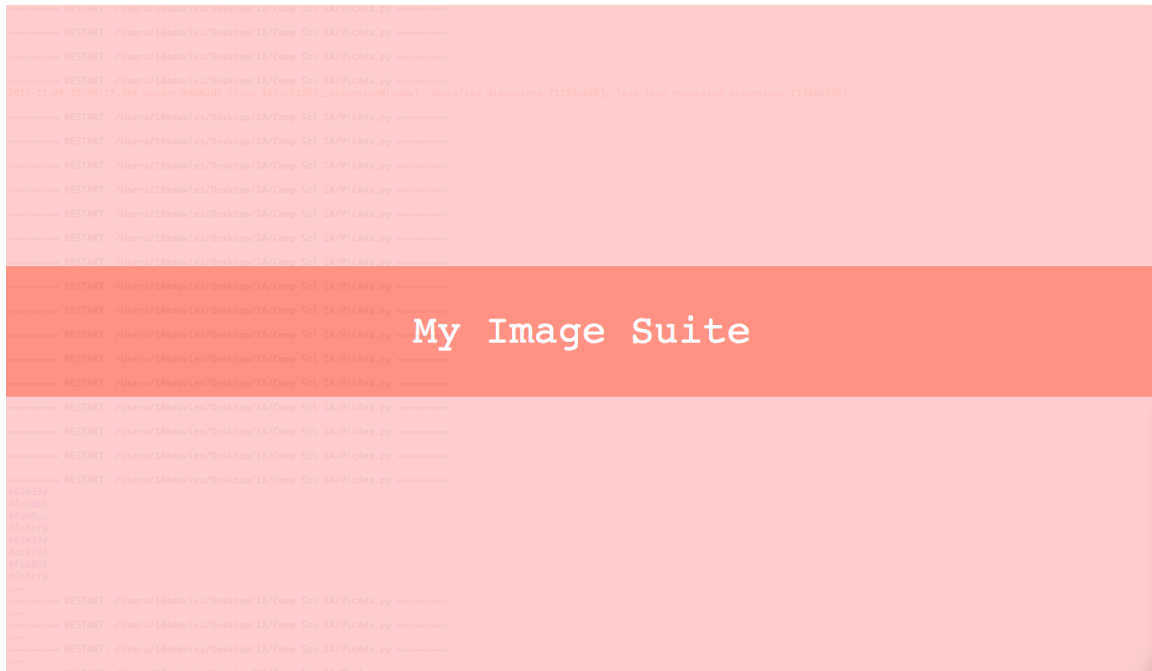
This product was originally developed as a command line application, which would be easier for programmers, but then I decided that a Graphical User Interface would ultimately be more user friendly for my client, who is a non-programmer, and other potential, non programmer clients.

Assisting Libraries

```
from PIL import Image
from colour import Color
from appJar import gui
import os
```

- Image module from Python Image Library (PIL, Pillow): This module provides a number of factory functions, including functions to load images from files, to modify, to create and to display new images. Once a user chooses a file to apply color changes to, using the gui module from the appJar Library, that chosen file is passed to the applyColor() function of this project, and edited by the Image module's various functions.
- Color Module from colour: It converts and manipulates the representation of common colors. It can also can pick colors for the user to identify objects of the application. It is mainly used in this project to identify colors by format and return a hexadecimal string representation of a color. It's an effective fail safe for utilizing colors in various methods without running into exceptions.
- gui module from appJar: The gui module from the appJar library allows programmers to construct and design GUI applications without dealing with too much boilerplate code, which is code that requires hundreds of thousands of lines to do mundane, minimalistic tasks.

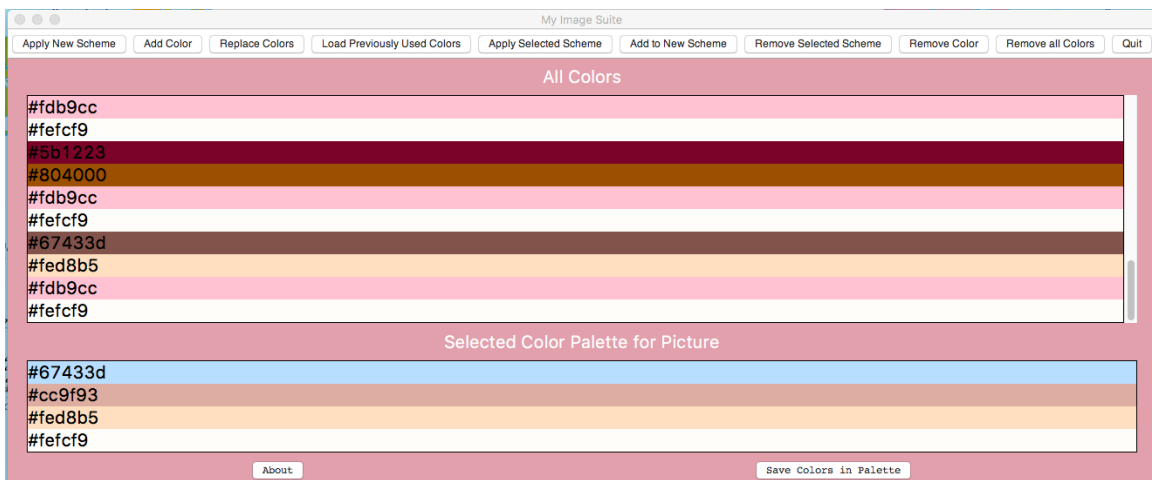
Graphical User Interface (GUI) Layout



This splash screen allows the program time to compile, and indicates to the user which software and what version they are launching.



This toolbar contains all of the buttons for the essential features of the application, such as adding colors and applying them to a photo. These buttons can be used to change settings or the functionality in a GUI. Toolbars appear across the top of a GUI.



Entire application window, complete with the toolbar, the log of colors, the button to save color schemes (bottom right) and the “About” button for the application (bottom left).

Techniques to Select and Format Colors

```
def getColors(myList="Colors"):
    sch = app.getListBox(myList)
    new_sch = []
    for i in sch:
        C1 = Color(i).rgb
        c1 = C1[0] * 255
        c2 = C1[1] * 255
        c3 = C1[2] * 255
        C1 = (int(c1), int(c2), int(c3))
        new_sch.append(C1)
    return new_sch
```

This function serves as a translator for the colors, from the user's input to a format that the picture can work with. It takes a list of the user's selected colors from any of the lists in the app (which by default is set to the general "Colors" list) and converts it to RGB format so that it may be applied to the pixels of a photo later on. The function returns a list of the inputted colors in their RGB format. First, the `rgb` property converts a color from hexadecimal form to a tuple of its percentage in float form (a decimal number between 0 to 1) of red, green and blue light. Because the percentage itself cannot be applied to the picture, each percentage is multiplied by 255 to get the true red, green and blue light values as a float. But because an image can only contain pixels with integers as the R, G and B values, that "true" value is converted to integer form with the `int(number)` function. This new tuple is then stored in a new scheme (`new_sch`) or list of colors, and that new list is returned by the function.

Techniques to Categorize and Apply Colors

```
def applyColor(im, scheme):
    colorpixels = list(im.getdata())
    list_length = len(colorpixels)
    for i in range(list_length):
        red = colorpixels[i][0]
        blue = colorpixels[i][1]
        green = colorpixels[i][2]
        sum = red + blue + green
        if sum < 175:
            colorpixels[i] = scheme[0]
        elif sum >= 175 and sum < 364:
            colorpixels[i] = scheme[1]
        elif sum >= 364 and sum < 546:
            colorpixels[i] = scheme[2]
        else:
            colorpixels[i] = scheme[3]
    im.putdata(colorpixels)
    im.show()
```

- Function `applyColor(im, scheme)`
 - `im`: The directory address of an image, instance of the Image module from the PIL.

- `scheme`: A list of colors that will be applied to the photo.

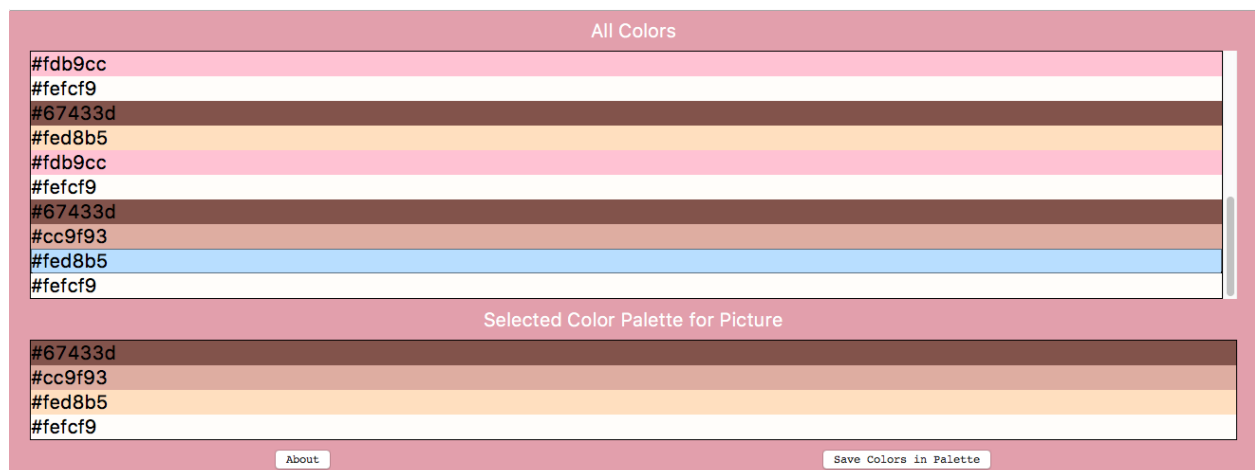
This function categorizes the image pixels by its total color intensity, which is the sum of the R, G and B values of the RGB pixel. With a range of 0 to 765, this measurement determines the total amount of color light value in a pixel. An intensity of 0 indicates that there is no red, green or blue light (i.e. no visible light) in the pixel. This pixel is classified as black. On the other side of the spectrum, an intensity of 765 indicates that there is the highest possible degree of visible light in the pixel. This pixel is classified as white. This function takes an image path (`im`) and a list of colors (`scheme`) as parameters. First, the pixels of the original picture are converted into an iterable list. Then, as the function parses through the list, each pixel in the image is separated into one of four color intensity-based categories. Then, based on that category and the user's selection of colors to apply, that pixel from the picture is replaced by a new tuple of R, G, B values, (i.e. a new color) from the scheme. The first color of the scheme is always the corresponding color for dark/black pixels, and the fourth color is the the corresponding color for light/white pixels. After the color changes are applied, the list of pixels is put into a new image using the `.putdata(list_of_pixels)` function of `Image` and displayed on the user's computer system's default viewer using the `.show()` function of `Image`.



The figure above is a screenshot of the “color palette” section of the application. From this list of colors, a new picture with these colors is generated. The user can also save their color scheme into a log, which can be accessed at any time.

Techniques to Store, Load and Remove Colors

```
def chooseColors():
    C1 = Color(app.colourBox())
    C1_RGB = C1.rgb
    c1 = C1_RGB[0] * 255
    c2 = C1_RGB[1] * 255
    c3 = C1_RGB[2] * 255
    C1_RGB = (int(c1), int(c2), int(c3))
    C2 = Color(app.colourBox())
    C2_RGB = C2.rgb
    c1 = C2_RGB[0] * 255
    c2 = C2_RGB[1] * 255
    c3 = C2_RGB[2] * 255
    C2_RGB = (int(c1), int(c2), int(c3))
    C3 = Color(app.colourBox())
    C3_RGB = C3.rgb
    c1 = C3_RGB[0] * 255
    c2 = C3_RGB[1] * 255
    c3 = C3_RGB[2] * 255
    C3_RGB = (int(c1), int(c2), int(c3))
    C4 = Color(app.colourBox())
    C4_RGB = C4.rgb
    c1 = C4_RGB[0] * 255
    c2 = C4_RGB[1] * 255
    c3 = C4_RGB[2] * 255
    C4_RGB = (int(c1), int(c2), int(c3))
    C = [C1_RGB, C2_RGB, C3_RGB, C4_RGB]
    C_RAW = [C1, C2, C3, C4]
    for i in C_RAW:
        if i == None:
            continue
        scheme.append(i)
        app.addListItem("Colors", str(i))
        app.setListItemBg("Colors", str(i), Color(i))
    return C
```

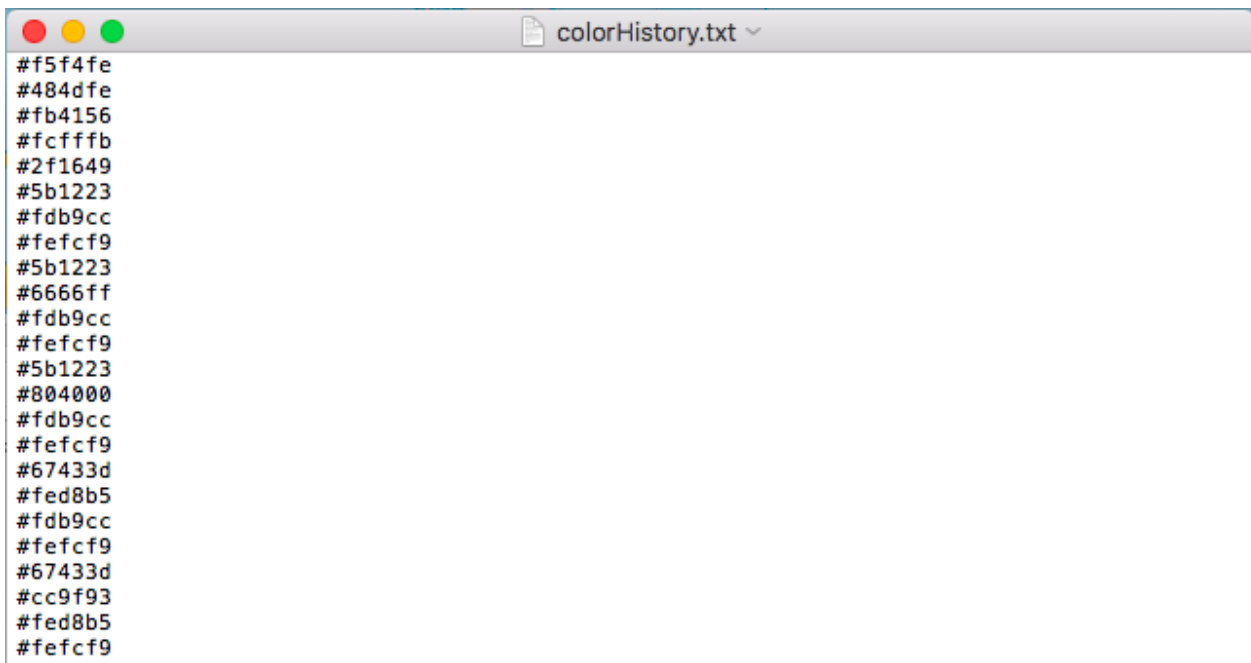


```
def saveColors(myList="Colors"):
    C = app.colourBox()
    if C == None:
        return
    scheme.append(C)
    app.addListItem(myList, str(C))
    app.setListItemBg(myList, str(C), C)
```

The user selects colors to add to their list and potentially apply to their new photo. Screenshot of the project code taken by the candidate. Rather than directly being added to the final list of selected colors for the picture, colors are, by default, added to a more general list of colors, called "All Colors", so that the user may filter through the unwanted colors and carefully select the colors that they want.

Save Colors in Palette

```
elif button == "Save Colors in Palette":
    selCC = app.getAllListItems("My Color Palette")
    if len(selCC) == 0:
        app.warningBox("Error Saving Colors", "Please add a color to the palette to continue.")
        return
    outFile = open ( "colorHistory.txt", "a" )
    for color in selCC:
        outFile.write ( color + '\n' )
        print(color)
    outFile.close ( )
```



A screenshot of a text editor window titled "colorHistory.txt". The window displays a list of 20 hex color codes, each on a new line. The colors are: #f5f4fe, #484dfe, #fb4156, #fcffff, #2f1649, #5b1223, #fdb9cc, #fefcf9, #5b1223, #6666ff, #fdb9cc, #fefcf9, #5b1223, #804000, #fdb9cc, #fefcf9, #67433d, #fed8b5, #fdb9cc, #fefcf9, #67433d, #cc9f93, #fed8b5, and #fefcf9.

```
#f5f4fe
#484dfe
#fb4156
#fcffff
#2f1649
#5b1223
#fdb9cc
#fefcf9
#5b1223
#6666ff
#fdb9cc
#fefcf9
#5b1223
#804000
#fdb9cc
#fefcf9
#67433d
#fed8b5
#fdb9cc
#fefcf9
#67433d
#cc9f93
#fed8b5
#fefcf9
```

```

elif button == "Load Previously Used Colors":
    with open ("colorHistory.txt", "r") as inFile :
        # read the file into data
        data = inFile.read ( )
        # make a list of lines
        myList = data.splitlines ( )
        for i in range(len(myList)):
            app.addListItem("Colors", str(myList[i]).replace('\n', ''))
            app.setListItemBg("Colors", str(myList[i]).replace('\n', ''), myList[i].replace('\n', ''))

```

Once the user has chosen a set of colors that they would like to have readily available at all times, they can save it for later use by pressing the “Save Colors in Palette” button, located under the list of the palette’s colors. Provided that the color palette has more than one color in it, this function saves the hex values of the colors by writing them to the file “colorHistory.txt”. When the user wants to re import the saved colors, the “Load Previously Used Colors” button on the toolbox reads all of the saved colors from the “colorHistory.txt” file and reloads them into the “All Colors” section of the application.



Remove Selected Scheme Remove Color Remove all Colors Replace Colors

```

elif button == "Replace Colors":
    selCC = app.getListBox("My Color Palette")
    selColor = app.getListBox("Colors")
    for item in selCC:
        color = app.colourBox()
        app.setListItem("My Color Palette", item, color)
        app.setListItemBg("My Color Palette", color, Color(color))
    for item in selColor:
        color = app.colourBox()
        app.setListItem("Colors", item, color)
        app.setListItemBg("Colors", color, Color(color))

elif button == "Remove Selected Scheme":
    app.clearListBox("My Color Palette")
elif button == "Remove Color":
    selMain = app.getListBox("Colors")
    selCC = app.getListBox("My Color Palette")
    for item in selMain:
        app.removeListItem("Colors", item)
    for item in selCC:
        app.removeListItem("My Color Palette", item)
elif button == "Remove all Colors":
    ans = app.okBox("Warning", "All colors will be permanently erased. Proceed?")
    if ans == True:
        app.clearAllListBoxes()

```

If the user does add one or more color to their scheme and then decides to change or remove that color, they have two options: remove or replace. These options result in actions that have different consequences, and it is up to the user to decide which feature is most appropriate for their individual situation.

- 'Remove' can remove colors in the app in three scales. The program can either remove some specifically highlighted colors, erase all the colors from the scheme and not the program, or erase all logged colors. Note that a color can still be re accessed through the "Load Previously Used Colors" button and its associated function.
- 'Replace' will swap one or more color(s) for another one, which the user will be prompted to choose a color for.

Results (Generated Pictures)

Before:



After:



This is an example of the end product of the user applying a color scheme to the photo. This example shows how the color intensity helped the application categorize each pixel, and will as identify areas of highlights and depth in the photo. The new colors have a rugged look that makes the picture look more like a painting.