Assignment 3: Color Models

(Source: Dept. of C.S., University of Cornell)

Submission deadline: February 26, 2023

This is an interesting assignment that takes advantage of the graphic capabilities of Python. However, since you have not yet been introduced to graphics in python, the code pertaining to graphics is being provided to you. You would only be required to implement functions based on whatever you have learnt thus far. One of the main things that you will learn in this assignment is that there are many different ways to represent color, and the choice of color model often depends on the application. For example, RGB is used when the colors need to be displayed on a computer monitor (such as a web site), while CMYK is often used for printing out colors on paper.

Important: Python has a built-in color conversion module called `colorsys`. You are forbidden from using that module in this assignment.

Learning Objectives

This assignment is designed to help you understand the following concepts.

- It introduces three color models that are used in computing and graphics.
- It gives you practice in writing complex functions with conditionals.
- It gives you practice with both fruitful and mutable functions.
- It introduces you to the notion of *attribute invariants.*
- It demonstrates a complex Python application that spans multiple modules.

Even though this is a complex Python application, most of the modules have been provided for you. You only need to focus on one module: `a3.py` (as well as the test script `a3test.py`).

Academic Integrity and Collaboration

Academic Integrity

- Do not post your code to any publicly accessible site.
- Do not indulge in plagiarism, we have zero tolerance for it.
- Python has a built-in color conversion module called `colorsys`. You are forbidden from using that module in this assignment.

Collaboration Policy

You may do this assignment with one other person. However, partner with someone only if required, you are encouraged to attempt it individually. If you do this assignment with another person, you must work together. It is against the rules for one person to do some programming on this assignment without the other person sitting nearby and helping. You are required not to look at anyone else's code or show your code to anyone else (besides your partner) in any form whatsoever. Also, be assured that **your score would not depend on whether or not you worked individually**.

Introduction to Color Models

Color Model RGB

The RGB system is named after the initials of the three color names: **red**, **green**, and **blue**. In this color model, light from these three colors is mixed to produce other colors, as shown in the image to the left. Black is the absence of color; white the maximum presence of all three.

In the upper right is a colored image. Below it is its separation into red, green, and blue (here is a **high resolution version**). In the three separation panels, the closer to black a point is, the less of that color it has. For example, the white snow is made up of a large amount of all three colors, whereas the brown barn is made up of red and green with very little blue. Because it works by adding colors to black, the RGB system is *additive*.

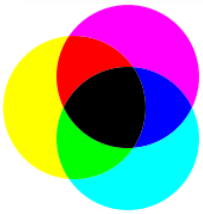
The color model RGB is used in your TV and computer screens, and hence on web pages. Its roots are in the 1953 RCA color-TV standards. However, the idea has been around longer. See **this exhibit** for some amazing full-color images taken with an RGB camera over 100 years ago.

In the RGB model used in most systems, the amount of red (R), green (G) and blue (B) is represented by a number in the range 0..255. Black, the absence of color, is [0, 0, 0].

White, the maximum presence of (R), (G), and (B), is [255, 255, 255]. This means that there are 16,777,216 different colors.

In some graphics systems, RGB is used with `float` numbers in the range 0.0..1.0 instead of int values 0..255. The reason for this discrepancy is that the mathematical formulas for color require real numbers 0.0..1.0, but it takes a lot less memory to store ints instead (and images require a lot of memory). In your program, you may have to convert each number in the integer range 0..255 to a `float` in 0.0..1.0, calculate a mathematical formula, and then convert back to 0..255.

Color Model CMYK



For your ink-jet printer, you buy expensive ink cartridges in the colors **cyan**, **magenta**, **yellow**, and **black**. The printer mixes these inks in different amounts on paper to make the full range of colors. Black is referred to using K (originally for "Key") to avoid confusion with Blue.

The process works similarly to RGB on a monitor, but in reverse. The paper starts off white (equal parts red, green, and blue), and the colors of these inks are chosen so that cyan ink absorbs red light, removing it from the color of the paper. Similarly, magenta removes green, and yellow removes blue. Black ink removes all three colors in equal amounts. For instance, paper printed with only yellow ink appears the same color as a monitor that is displaying a yellow color [255, 255, 0] because it has removed all the blue, leaving the red and green. Printing magenta and cyan removes red and yellow and results in blue [0, 0, 255]. Because it works by removing color, this kind of system is *subtractive*.

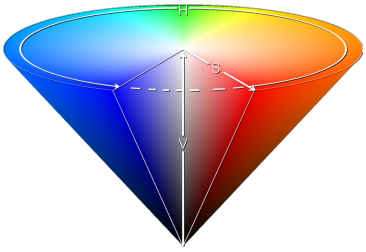
Theoretically, only C, M, and Y are needed to achieve any color, but in practice it is hard to get a good black by mixing colored inks. Instead you get a soggy, expensive brown-black. By using the black ink to do the “heavy lifting” of absorbing most of the

light when printing dark colors, a lot of ink can be saved (This is a simplified view of color printing; more complicated calculations are needed to get accurate colors with real inks).

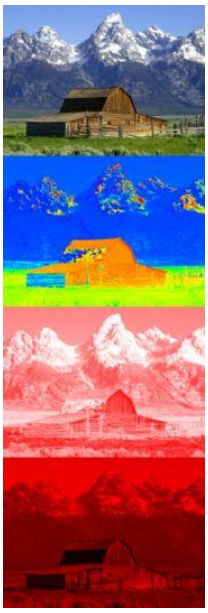
To demonstrate this issue, look at the image in the upper right. The left version of the image is its separation into cyan, magenta, and yellow (**enlarged version**). To the right of that, you see the same image separated into four components; C, M, Y, K (**enlarged version**). As you can see, much less CMY ink is needed to make the image when black is also used.

In the CMYK system, each of the four components is traditionally represented by a percentage. We will use `float` values in the range 0.0..100.0.

Color Model HSV



The HSV model, used heavily in graphics applications, was created in 1978 by Alvy Ray Smith. Artists prefer the HSV model over others because of its similarities to the way humans perceive color. HSV can be explained in terms of the cone that appears to the left. It is broken up into three values.



H, the *hue*, defines the basic color. H is an angle in the range 0 ≤ H < 360, if one views the top of the cone as a disk. Red is at angle 0. As the angle increases, the hue changes to orange, yellow, green, cyan, blue, violet, magenta, and back to red. The color wheel in the **ColorModel application** that you are developing would make this correlation explicit.

S, in the range 0 ≤ S ≤ 1, is the *saturation*. It indicates the distance from the center of the disk. The lower the S value, the more faded and grayer the color. The higher the S value, the stronger and more vibrant the color.

V is the *value* or *brightness*. It is also in the range 0 ≤ V ≤ 1. It indicates the distance along the line from the point of the cone to the disk at the top. If V is 0, the color is black; if 1, the color is as bright as possible.

To the right at the top is a picture. Below it we see its hue, saturation (white is zero saturation, red is full saturation), and brightness components. The hue component shows color. The snow has color, but its saturation is low, making it almost grayish. Look at the various components of the image —the sky, the green grass, the snow, the dark side of the barn, and so on to see how each component `H`, `S`, and `V` contributes. You can see more detail in this **high-resolution version**.

The `introcs` Color Classes

All of the color models of the previous section are provided by the module `introcs`, available with the code files for this assignment. This module provides three different classes: `RGB`, `CMYK`, and `HSV`. It lacks the ability to convert between these classes. That is the focus of this assignment.

The Class `RGB`

The class `RGB` is the type of objects that represent RGB color. Objects of type `RGB` have three attributes: `red`, `green`, and `blue` (They also have a secret attribute `alpha`, but this will not be used in this assignment). For example, if `c` is a variable containing a (name of) an RGB object, you would use the expression `c.red` to access the red value.

The `RGB` constructor function takes three arguments, assigning these values to the attributes in the order `red`, `green`, and `blue`. For example, to create an `RGB` object representing the color red, use the assignment

```
red = introcs.RGB(255,0,0)
```

The Class `CMYK`

The class `CMYK` is the type of objects that represent CMYK color. Objects of type `CMYK` have four attributes: `cyan`, `magenta`, `yellow`, and `black`. For example, if `c` is a variable containing a (name of) a CMYK object, you would use the expression `c.cyan` to access the cyan value.

The `CMYK` constructor function takes four arguments, assigning these values to the attributes in the order `cyan`, `magenta`, `yellow`, and `black`. For example, to create a `CMYK` object representing the color red, use the assignment

```
red = introcs.CMYK(0.0,100.0,100.0,0.0)
```

The Class `HSV`

The class `HSV` is the type of objects that represent HSV color. Objects of type `HSV` have three attributes: `hue` `saturation`, and `value`. For example, if `c` is a variable containing a (name of) a HSV object, you would use the expression `c.hue` to access the hue value.

The `HSV` constructor function takes three arguments, assigning these values to the attributes in the order `hue`, `saturation`, and `value`. For example, to create an `HSV` object representing the color red, use the assignment

```
red = introcs.HSV(0.0,1.0,1.0)
```

Attribute Invariants

All of the objects in this assignment have *attribute invariants*. An attribute invariant is a property of an attribute (which is essentially a variable) inside an object. The invariant cannot be violated. Attempting to violate an invariant will cause an error and crash Python.

For example, for RGB objects, the `red` attribute has an invariant that it must be an `int` and it must be in the range 0..255, inclusive. The following code produces an error:

```
>>> import introcs

>>> rgb = introcs.RGB(255,255,255)

>>> rgb.red = -1

Traceback (most recent call last):

  ...

ValueError: value -1 for attribute red is outside of range
[0,255]
```

The invariants in the introcs color classes are provided for your benefit. They are there to help you catch errors. All you need to do is to make sure that you never assign a value to an attribute that violates an invariant.

The invariants for this assignment are as follows:

- All attributes of `RGB` must be ints, and in the range 0 to 255, inclusive.
- All attributes of `CMYK` must be floats, and in the range 0.0 to 100.0, inclusive.
- The `hue` attribute of `HSV` must be a float in the range 0.0 to 360.0, not including 360.

- The `saturation` and `value` attributes must be floats in the range 0.0 to 1.0, inclusive.

The ColorModel Application

This assignment provides you with a lot of the code already written. In addition, you are being provided with an online **color conversion tool**, so that you know what your answers should look like.

Assignment Source Code

Download the source code and support files to this assignment before you do anything else. As said above, this assignment will involve several files. Two are already complete, and the other two have function stubs or incomplete implementations that you must finish yourselves. **They must all be in the same directory for this assignment to work.**

The following are the two completed source code files (**You should not need to modify the contents of any of these files at all**):

- `a3app.py`: The Kivy script providing the GUI application.
- `colormodel.kv`: A layout file used to arrange the input controls in the GUI window.

You should not expect to understand the code in `a3app.py`. As long as you follow the directions of the assignment, you should be able to run this script without understanding it. In fact, it is not necessary to run this script to complete this assignment, but it does make it a little more fun.

In addition to those two files, the following source code files are skeletons (i.e. they are incomplete and you are expected to add functionality to them):

- `a3.py`: The module with the functions that you are to implement.
- `a3test.py`: The test script for the functions in `a3.py`.

The file `a3.py` is completely unfinished. It has nothing but stubs. However, to make the assignment easier, many of the test cases have been provided for you already in `a3test.py`. With that said, not all test procedures are complete, and you should add more tests as the instructions tell you to do so. You have the following functions implemented in the introcs module that you may use to write test procedures:

assert_equals

introcs.assert_equals(*expected*, *received*)

Quits if `expected` and `received` differ.

The meaning of "differ" for this function is !=. As a result, this assert function is not necessarily reliable when expected and received are of type `float`. You should use the function **`assert_floats_equal()`** for that application.

This function will print some minimal debug information. The following is an example debug message:

```
assert_equals: expected 'yes' but instead got 'no'
```

Parameters

- **expected** – The value you expect the test to have
- **received** – The value the test actually had

assert_floats_equal

introcs.assert_floats_equal(*expected*, *received*)

Quits if the floats `expected` and `received` differ.

This function takes two numbers and compares them using functions from the numerical package `numpy`. This is a scientific computing package that allows us to test if numbers are "close enough". Hence, unlike **`assert_equal()`**, the meaning of "differ" for this function is defined by numpy.

This function will print some minimal debug information. The following is an example debug message:

```
assert_floats_equal: expected 0.1 but instead got 0.2
```

IMPORTANT: The arguments expected and received should both be numbers (either floats or ints). If either argument is not a number, the function quits with a different error message. For example:

```
assert_floats_equal: The first argument is not a number
```

Parameters

- **expected** (`float`) – The value you expect the test to have
- **received** (`float`) – The value the test actually had

Technically, you can complete this assignment with just `a3.py` and `a3test.py`. As said earlier, the GUI is just intended to make the assignment a little more fun.

Understanding the GUI

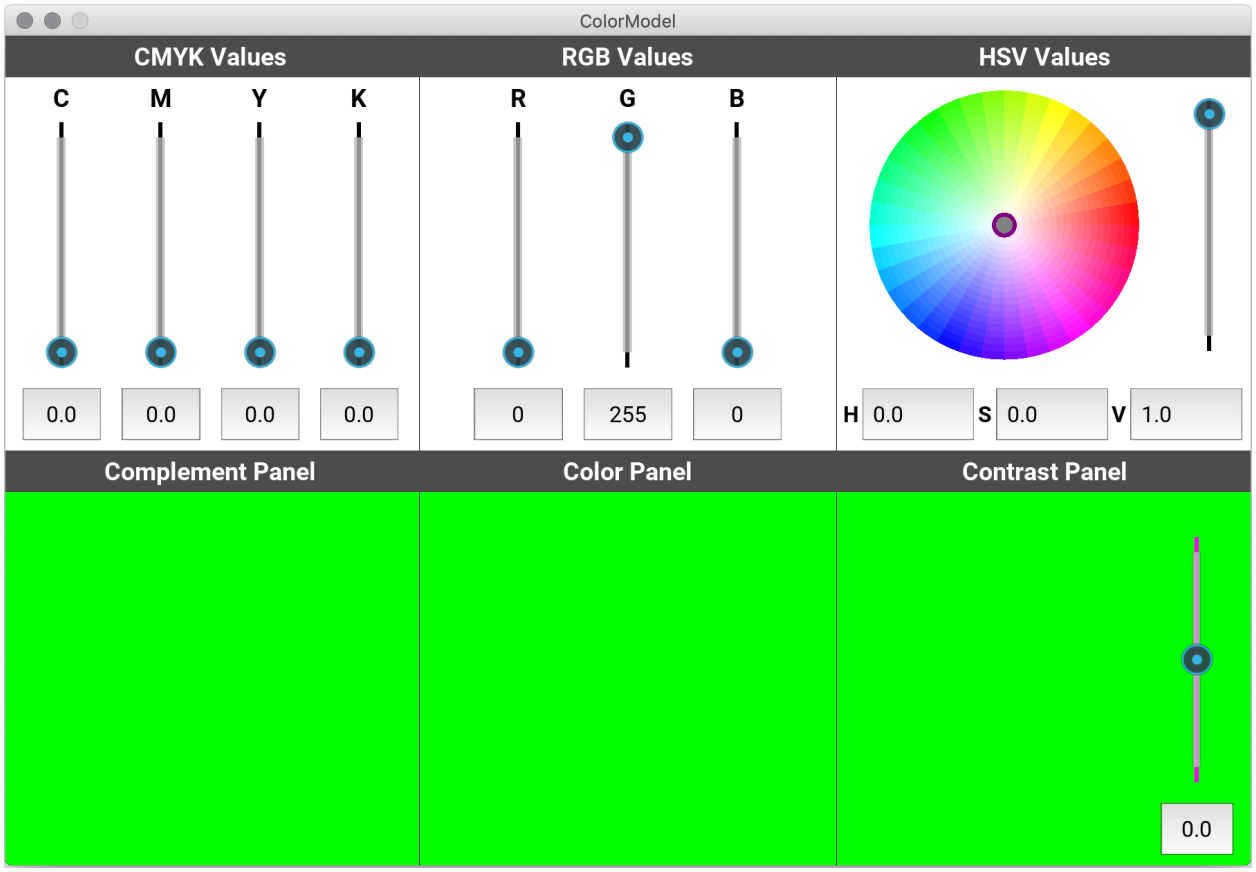
To run the GUI, make sure all four files are in the same directory. Then navigate to this directory and run `a3app.py` as a script as follows:

```
   python a3app.py
```

You will see a bunch of crazy messages that look like this:

```
[INFO   ] [Logger      ] Record log in ...
[INFO   ] [Kivy        ] v1.11.1 ...
[INFO   ] [Factory     ] 184 symbols loaded
[INFO   ] [Image       ] Providers: img_tex, img_imageio, ...
[INFO   ] [Text        ] Provider: sdl2
[INFO   ] [Window      ] Provider: sdl2
[INFO   ] [GL          ] Using the "OpenGL ES 2" graphics system
...
```

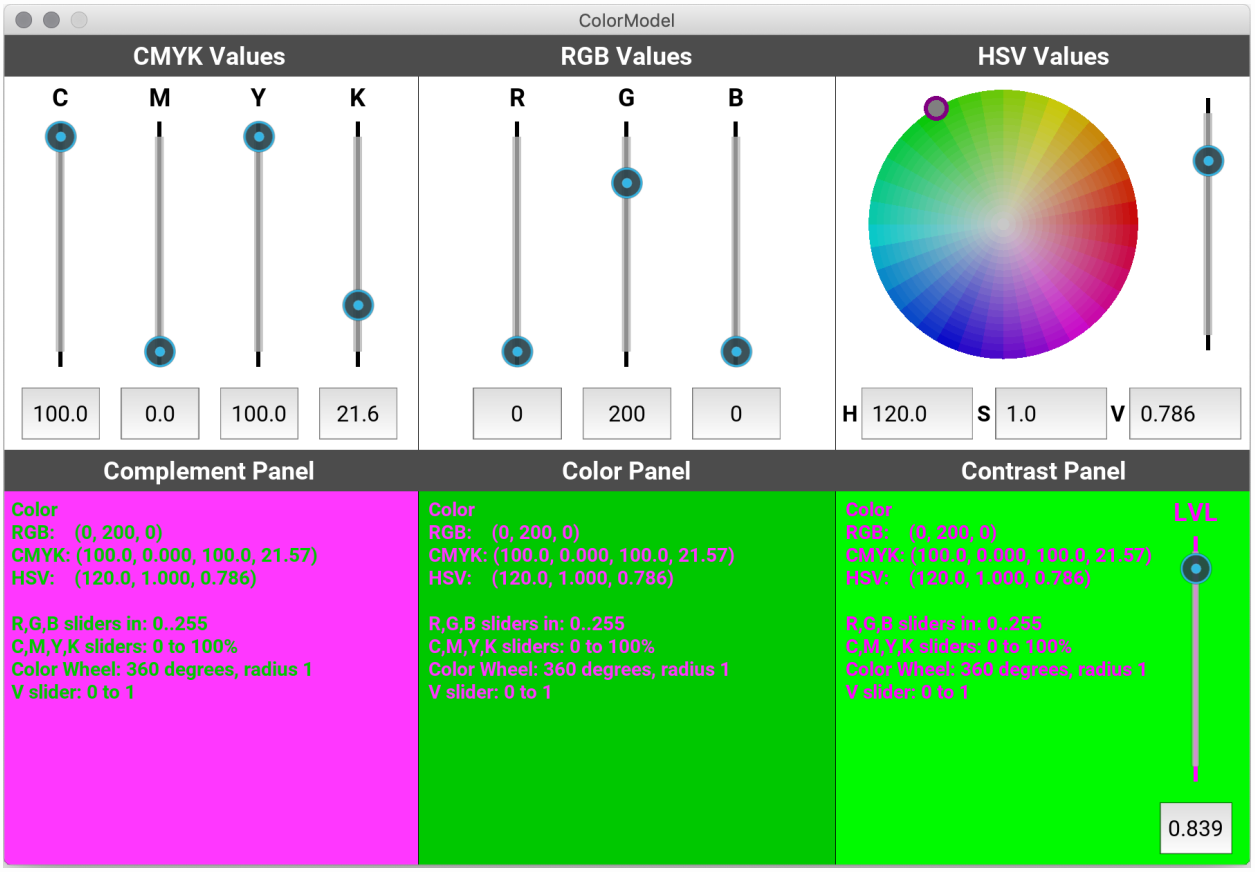
That is Kivy (our GUI library) initializing the application. When the messages are done, you should see a GUI window that looks like the window shown below.



In this application, you see some sliders (and a color wheel) up top, and some colored panels down at the bottom. Right now, very little of it works. If you move the RGB sliders, you will see the color panels change color. However, none of the other controls work. You can also change the color by entering the new color value into the R, G, and B fields.

Your job is to write and test, *one by one*, the functions in module `a3`. As you do, more and more of the GUI will work properly.

A working GUI should look like the one shown below. The first two color panels at the bottom should be two different colors, and they should each be the complement of the other. The text will also be in the complement color, and it will display the color values, in RGB, CMYK, and HSV, for the background color. The right color panel will have a contrast setting that allows you to make the central color brighter or darker.



You can change these colors by moving the controls. If you move the controls in one color model, then the controls in the other color models will follow automatically. This way, all three models (RGB, CMYK, and HSV) register the same color. The contrast control is completely separate and disconnected, which is why it is at the bottom.

The numbers in the text field should be in their respective ranges: 0..255 for R, G, B; 0.0..100.0 for C, M, Y, K; 0.0..1.0 for S, V; and 0.0..359.999 for H. The CMYK numbers will display to one decimal place, while the HSV numbers will display to 3 decimal places (the RGB numbers are integers). If you type in something that is not a string, or a number out of range, the input will be ignored and it will revert back to the previous value.

Experimenting with the Color Models

You may use the provided **color conversion tool** to get an idea of what the answers should look like for various inputs (after you enter a value in a field on that page, you have to click on a different field to get the whole page to update to the new values). For CMYK and HSV the answers provided are **accurate to three decimals places**. We do not guarantee more accuracy than that. You should use this page to help you design your test cases for the rest of the assignment.

Testing and Debugging

For testing, you would be required to use the script `a3test.py`; several of the test procedures have been started for you. **You will be graded on the completeness of your test cases**.

As you debug your code, you may want to add print statements to `a3.py` as watches (e.g. print statements that display the contents of a variable) and traces (e.g. print statements that indicate the line of code that is currently executing). Traces will be particularly valuable for this assignment because it is the first major assignment involving conditionals.

Because this is a complex assignment, we recommend that you be very descriptive with your watches and traces. Suppose you are trying to find an error in the function `rgb_to_hsv` (or in a function that calls `rgb_to_hsv`) and that `rgb_to_hsv` changes a variable `h` at line 170. Then, you might insert at line 171 a statement like

```
    print('rgb_to_hsv: h at line 171 is '+ str(h))
```

Assignment Instructions

In this section we outline the functions that you are expected to write for this assignment. You should write **and test** your functions, one at a time, in the order given. Read the text below, but also make sure you carefully read the specifications and comments in the provided code.

Function `complement_rgb`

The first function is a warm-up. We gave you some partial code that you need to fix. The complement of a color is like a color negative. If R, G, and B were color components of the RGB value in the range 0.0..1.0 (not 0..255!), then the color components of the complement would be 1-R, 1-G, and 1-B. However, since we are using values in the range 0..255, the complementary color of the RGB color `(r, g, b)` is the color `(255-r, 255-g, 255-b)`.

Currently this function does not work. Instead, it makes a copy of the RGB object in the parameter variable. You will need to change the arguments to the RGB constructor to get it to return the complement instead. A complete test procedure has been given in `a3test.py` for you to verify that your function implementation is correct.

After completing this function, run `a3app.py` as a script. You will now see text in the colored boxes. This text will have a lot of information, including the RGB value of the current color. However, the CMYK and HSV information will still be blank.

The `str5` Functions

To turn a color object into a string, you can use the `str` function which you have seen in class. This is fine for RGB objects, as the attributes are integers. However, CMYK and HSV objects have float attributes, and they are a lot messier. Floats could potentially have 18 digits!

To solve this problem, we want you to create some functions that limit all of the numbers to five characters (not digits). The following example shows the difference between `str` and `str5_hsv`:

```
>>> import introcs

>>> color = introcs.HSV(128.54,0.46792,0.32456)

>>> str(color)

'(128.54,0.46792,0.32456)'

>>> str5_hsv(color)

'(128.5, 0.468, 0.325)'
```

`str5`

This function simply rounds a number and converts it to a string. The challenge is that the number of places to round to is not constant. For example, `str5(1.0567)` returns `'1.057'`, while `str5(10.567)` returns `'10.57'`. When implementing this function you should pay close attention to the precondition (as it will be helpful).

Within this assignment, this function should *only* be used by the GUI to give it a consistent format. You should never use this function anywhere in your code other than `str5_cmyk` or `str5_hsv`. In particular, you should never use it in a conversion function, since that results in a loss of mathematical precision.

`str5_cmyk`

Implement this function according to the specification in `a3.py`. This function should call function `str5` to round each CMYK value to 5 characters. We have provided you with two test cases in `a3test.py` to show you how to test this function. You are welcome to add more, but this is not necessary.

`str5_hsv`

Implement this function according to the specification in `a3.py`. This function should call function `str5` to round each CMYK value to 5 characters. We do **not** provide test cases for this function. You must write at least two of them.

Test Cases

As we mentioned above, the tests for `str5` and `str5_cmyk` are complete. You are welcome to add more, but this is not necessary. However, there are no tests for `str5_hsv`. We expect you to add at least two. They go in the test procedure `test_str5_color`. Look at the tests for `str5_cmyk` as an example for how to create these tests.

The CMYK Functions

The next two functions convert back-and-forth between RGB and CMYK colors. When you implement the function `rgb_to_cmyk`, the numerical CMYK color will display properly in the lower color panes (assuming that `str5_cmyk` is implemented correctly). In addition, moving the RGB sliders will cause the CMYK sliders to move as well. However, the reverse will not be true until you finish `cmyk_to_rgb`. There will also be no effect to the HSV controls.

When you complete the second function `cmyk_to_rgb`, moving the CMYK sliders will cause the RGB sliders to also move. At this point, the two sets of sliders will work in tandem.

`rgb_to_cmyk`

This function converts an RGB value to a CMYK value. There are several different ways to convert, depending on how much black is used in the CMYK model. Our conversion uses as much black as possible. Let R, G, and B be the color components of the RGB value in the range 0.0..1.0 (not 0..255!). That means that you will need to *first divide the*

values in the RGB object by 255.0. Once you do that, then the conversion is as follows. First compute

$$K = 1 - \max(R, G, B)$$

If K is 1, then the other color (C, M, Y) values are all 0. Otherwise, compute them with the following formulas:

$$C = (1 - R - K)/(1 - K)$$
$$M = (1 - G - K)/(1 - K)$$
$$Y = (1 - B - K)/(1 - K)$$

The resulting CMYK values are in the range 0.0..1.0, and they must be converted to the range 0..100.0. And that is it! Not too bad, right? However, **do not round your answers**. That is an unacceptable loss of precision.

Important: While we use capital letters in the mathematical formulas above, do not use capital letters as variables. Make them lower-case in your code.

`cmyk_to_rgb`

This function converts a CMYK value to an RGB value. Let C, M, Y, and K be the color components of the CMYK value, all in the range 0.0..1.0 (not 0..100.0; you will need to convert this first). Then the conversion is as follows:

$$R = (1 - C)(1 - K)$$
$$G = (1 - M)(1 - K)$$
$$B = (1 - Y)(1 - K)$$

This produces RGB values in the range 0.0..1.0, and they must be converted to the range 0..255. You should *use rounding* in converting the answer to an int. Remember: **ROUND; DO NOT TRUNCATE**. Remember that casting truncates, and does not round.

Test Cases

The test cases for `rgb_to_cmyk` have already been provided. However, you should study them as you will need to emulate them when you work on the HSV functions. Test cases for CMYK values are difficult because the `float` attributes are only approximations to the real values, and slightly different ways of computing might produce different results. Instead of using `assert_floats_equal`, we round each attribute to three decimal places (in the *test*, not the *function*). That is because this is the degree of accuracy we are requiring. Your answers only need to be correct to three decimal places.

For `cmyk_to_rgb` we expect you to create your own tests. This is straight-forward, since the attributes of RGB are all integers. We will not tell you how many tests that you need, but you do need to have proper *code coverage*. Do you have an if-statement in the code for `cmyk_to_rgb`? If so, you need to make sure that you have a test for each possible branch of the if.

In picking color values to test, you should make use of the **online color converter**. This tool is accurate up to three decimal places.

The HSV Functions

The next two functions convert back-and-forth between RGB and HSV colors. When you implement the function `rgb_to_hsv`, the numerical HSV color will display properly in the lower color panes (assuming that `str5_hsv` is implemented correctly). In addition, moving the RGB sliders will cause the HSV color wheel to move as well. However, the reverse will not be true until you finish `hsv_to_rgb`.

When you complete the second function `hsv_to_rgb`, moving the color wheel will cause the RGB sliders to also move. At this point, all of the upper level controls should work together.

`rgb_to_hsv`

This color conversion is a little more complicated that the previous ones. First, convert the RGB values so that R, G, and B are in the range 0..1. Next let *M* be the **maximum** and *m* be the **minimum** of the (R, G, B) values.

The value H is now given by 5 different cases:

$$H = \begin{cases} 0 & \text{if } M = m \\ 60.0 * (G - B)/(M - m) & \text{if } M = R, G \geq B \\ 60.0 * (G - B)/(M - m) + 360.0 & \text{if } M = R, G < B \\ 60.0 * (B - R)/(M - m) + 120.0 & \text{if } M = G \\ 60.0 * (R - G)/(M - m) + 240.0 & \text{if } M = B \end{cases}$$

The value S is given by the formula

$$S = \begin{cases} 0 & \text{if } M = 0 \\ 1 - m/M & \text{otherwise} \end{cases}$$

Finally, V = *M*. **Do not round your answers**. That is an unacceptable loss of precision.

`hsv_to_rgb`

This function converts an HSV value to an RGB value. To perform the conversion, you first need to compute the following values.

$$
\begin{aligned}
H_i &= \text{floor}(H/60) \\
f &= H/60 - H_i \\
p &= V(1-S) \\
q &= V(1-fS) \\
t &= V(1-(1-f)S)
\end{aligned}
$$

Once you have this computed, the values R, G, and B depend on the value H_i as follows:

$$
R = \begin{cases}
V & \text{if } H_i = 0, 5 \\
q & \text{if } H_i = 1 \\
p & \text{if } H_i = 2, 3 \\
t & \text{if } H_i = 4
\end{cases}
$$

$$
G = \begin{cases}
t & \text{if } H_i = 0 \\
V & \text{if } H_i = 1, 2 \\
q & \text{if } H_i = 3 \\
p & \text{if } H_i = 4, 5
\end{cases}
$$

$$
B = \begin{cases}
p & \text{if } H_i = 0, 1 \\
t & \text{if } H_i = 2 \\
V & \text{if } H_i = 3, 4 \\
q & \text{if } H_i = 5
\end{cases}
$$

This produces RGB values in the range 0.0..1.0, and they must be converted to the range 0..255. You should *use rounding* in converting the answer to an `int`. Remember: **ROUND; DO NOT TRUNCATE**. Remember that casting truncates, and does not round.

Test Cases

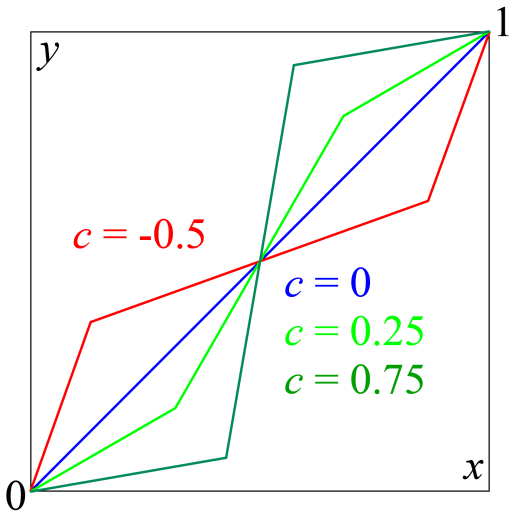
We have not provided you with any tests for either `rgb_to_hsv` or `hsv_to_rgb`. You must supply the tests for both. Once again, you must have proper *code coverage*. Do you have an if-statement in either function? If so, you need to make sure that you have a test for each possible branch of the if.

For the tests for `rgb_to_hsv`, you only need to test that the `hue`, `saturation`, and `value` attributes are accurate up to three decimal places. To help with this test, you should make use of the **online color converter**, which is also accurate up to three decimal places.

The Contrast Functions

The last two functions implement the contrast slider in the bottom right-hand corner of the GUI application. A *contrast value* is a number between -1 and 1. At 0 contrast, all colors are left untouched. For a positive contrast, bright colors are made brighter and dark colors are made darker. For a negative contrast, colors come closer together and are harder to tell apart. This is similar to how a contrast setting works on a television or a computer monitor.

There are many ways to implement a contrast slider. We are going to use a *sawtooth curve*.



This curve is the line $y = x$ (between 0 and 1) when the contrast c is 0. As c approaches -1, it becomes the horizontal line $y = 0.5$. As c approaches 1, it splits into two horizontal lines at 0 and 1.

More formally, for $-1 \leq c < 1$, we define this curve as

$$y = \begin{cases} \left(\frac{1-c}{1+c}\right)x & \text{if } x < 0.25 + 0.25c \\ \left(\frac{1-c}{1+c}\right)\left(x - \frac{3-c}{4}\right) + \frac{3+c}{4} & \text{if } x > 0.75 - 0.25c \\ \left(\frac{1+c}{1-c}\right)\left(x - \frac{1+c}{4}\right) + \frac{1-c}{4} & \text{otherwise} \end{cases}$$

In the case where $c = 1$, it becomes the discontinuous curve

$$y = \begin{cases} 1 & \text{if } x \geq 0.5 \\ 0 & \text{otherwise} \end{cases}$$

`contrast_value`

To implement this function, simply follow the formula above. The parameter `value` is x and the parameter `contrast` is c. The return value is y. There are no color objects involved in this function; only numbers.

`contrast_rgb`

Note that this function is a *mutable function*. It does not return a new RGB object. Instead, it modifies the RGB object that was passed as an argument. If you look at the test in the procedure `test_contrast_rgb` of `a3test.py` you can get some idea of what it should do.

To apply contrast to an RGB object, first convert the values R, G, B to the range 0.0..1.0. Then apply the function `contrast_value` above to get new values for R, G, B. Finally convert the result back to 0..255 using multiplication and rounding.

Test Cases

The sawtooth curve has a lot of possibilities, and so it requires a lot of test cases. For that reason, all of the tests for `contrast_value` have been provided already. However, only one test for `contrast_rgb` has been provided; you are required to provide at least two more tests. Use the tests in `test_contrast_value` as a guide for how to come up with tests for this function.

Submission

Before you submit this assignment, you should be sure that everything is working and polished. check the following before submitting:

- Functions are each separated by two blank lines.
- The specifications for all of the functions are complete.
- Function specifications are immediately after the function header and indented.
- Docstrings are only used for specifications, not general comments.
- Write your name(s) at the top of the modules to be submitted.

Save the files a3.py and a3test.py in a folder, name the folder with your name(s) and upload the folder here by the **due date: Sunday, February 26, 2023**. No other files are to be uploaded.

Submission would be **followed by a viva**. Please note that even those who attempt the assignment in pairs would have to appear for the viva **individually**.