Colorzero 2.0 Documentation

Release 2.0

Dave Jones

Nov 14, 2021

CONTENTS

CHAPTER

ONE

INSTALLATION

1.1 Debian installation

On Debian[1] (including its derivatives, Ubuntu[2] and RaspiOS[3]) it is best to obtain colorzero via the `apt` utility:

```
$ sudo apt update
$ sudo apt install python3-colorzero
```

The usual apt upgrade method can be used to keep your installation up to date:

```
$ sudo apt update
$ sudo apt upgrade
```

To remove your installation:

```
$ sudo apt remove python3-colorzero
```

1.2 Other platforms

On other platforms, it is probably easiest to obtain colorzero via the `pip` utility:

```
$ sudo pip3 install colorzero
```

To upgrade your installation:

```
$ sudo pip3 install -U colorzero
```

To remove your installation:

```
$ sudo pip3 remove colorzero
```

[1] https://www.debian.org/
[2] https://ubuntu.com/
[3] https://www.raspberrypi.com/software/

CHAPTER

TWO

GETTING STARTED

The *Color* (page 7) class is the main interface provided by colorzero. It can be constructed in a large variety of ways including with red, green, and blue components, "well known" color names (taken from CSS 3's extended color keywords[4]), HTML color specifications, and more. A selection of valid constructors is shown below:

```
>>> from colorzero import *
>>> Color('red')
<Color html="#ff0000" rgb=(1.0, 0.0, 0.0)>
>>> Color(1.0, 0.0, 0.0)
<Color html="#ff0000" rgb=(1.0, 0.0, 0.0)>
>>> Color(255, 0, 0)
<Color html="#ff0000" rgb=(1.0, 0.0, 0.0)>
>>> Color('#ff0000')
<Color html="#ff0000" rgb=(1.0, 0.0, 0.0)>
>>> Color('#f00')
<Color html="#ff0000" rgb=(1.0, 0.0, 0.0)>
```

Internally, colorzero always represents colors as red, green, and blue values between 0.0 and 1.0. *Color* (page 7) objects are tuple descendents. Crucially, this means they are *immutable*. Attempting to change the red, green, or blue attributes will fail:

```
>>> c = Color('red')
>>> c.red
Red(1.0)
>>> c.red = 0.5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```

In order to manipulate a color, colorzero provides a simple series of classes which represent attributes of a color: *Red* (page 15), *Green* (page 16), *Blue* (page 16), *Hue* (page 16), *Lightness* (page 17), *Saturation* (page 16) and so on. You can use these classes in combination with Python's usual mathematical operators (addition, subtraction, multiplication, etc.) to manipulate a color. For example, continuing the example from above:

```
>>> c + Green(0.1)
<Color html='#ff1a00' rgb=(1, 0.1, 0)>
>>> c = c + Green(0.5)
>>> c
<Color html='#ff8000' rgb=(1, 0.5, 0)>
>>> c.lightness
Lightness(0.5)
>>> c = c * Lightness(0.5)
>>> c
<Color html='#804000' rgb=(0.5, 0.25, 0)>
```

Numerous attributes are provided to enable conversion of the RGB representation to other systems:

[4] https://www.w3.org/TR/css3-color/#svg-color

```
>>> c.rgb
RGB(r=0.5, g=0.25, b=0.0)
>>> c.rgb_bytes
RGB(r=128, g=64, b=0)
>>> c.rgb565
31200
>>> c.hls
HLS(h=0.08333333333333333, l=0.25, s=1.0)
>>> c.xyz
XYZ(x=0.10647471144683732, y=0.0819048964489466, z=0.010202272707313633)
>>> c.lab
Lab(l=34.376494620040376, a=23.890819210881016, b=44.69197916172735)
```

Equivalent constructors exist for all these systems:

```
>>> Color.from_rgb(0.5, 0.25, 0.0)
<Color html='#804000' rgb=(0.5, 0.25, 0)>
>>> Color.from_rgb_bytes(128, 64, 0)
<Color html='#804000' rgb=(0.501961, 0.25098, 0)>
>>> Color.from_rgb565(31200)
<Color html='#7b3d00' rgb=(0.483871, 0.238095, 0)>
>>> Color.from_hls(*c.hls)
<Color html='#804000' rgb=(0.5, 0.25, 0)>
>>> Color.from_xyz(*c.xyz)
<Color html='#7f4000' rgb=(0.5, 0.25, 0)>
>>> Color.from_lab(*c.lab)
<Color html='#7f4000' rgb=(0.5, 0.25, 0)>
```

Note that some conversions lose a certain amount of precision.

The `repr()`[5] output of *`Color`* (page 7) is relatively verbose by default, but this can be customized via the *`Color.repr_style`* (page 9) class attribute:

```
>>> c = Color('red')
>>> c
<Color html="#ff0000" rgb=(1.0, 0.0, 0.0)>
>>> Color.repr_style = 'html'
>>> c
Color('#ff0000')
>>> Color.repr_style = 'rgb'
>>> c
Color(1, 0, 0)
```

If you have a terminal capable of color output (usually this means an actual terminal, not those integrated into applications like IDLE, Thonny, etc.), you can also preview colors with this facility (the output below shows the ANSI codes produced, but the documentation system won't reproduce the colored output):

```
>>> Color.repr_style = 'term256'
>>> c
<Color ### rgb=(1, 0, 0)>
>>> repr(c)
'<Color \x1b[38;5;9m###\x1b[0m rgb=(1, 0, 0)>'
>>> Color.repr_style = 'term16m'
>>> c
<Color ### rgb=(1, 0, 0)>
>>> repr(c)
'<Color \x1b[38;2;255;0;0m###\x1b[0m rgb=(1, 0, 0)>'
```

These ANSI codes can also be generated by using colors with `str.format()`[6]. For example:

[5] https://docs.python.org/3.9/library/functions.html#repr
[6] https://docs.python.org/3.9/library/stdtypes.html#str.format

```
>>> '{c:16m}Red{c:0} Alert!'.format(c=Color('red'))
'\x1b[38;2;255;0;0mRed\x1b[0m Alert!'
```

See *Format Strings* (page 14) for more information.

A method (*gradient()* (page 11)) is provided to generate gradients which fade from one color to another. The result is a generator, which must be iterated over if you want all the results:

```
>>> Color.repr_style = 'term16m'
>>> for c in Color('red').gradient(Color('green')):
...     print(repr(c))
...
<Color ### rgb=(1, 0, 0)>
<Color ### rgb=(0.888889, 0.0557734, 0)>
<Color ### rgb=(0.777778, 0.111547, 0)>
<Color ### rgb=(0.666667, 0.16732, 0)>
<Color ### rgb=(0.555556, 0.223094, 0)>
<Color ### rgb=(0.444444, 0.278867, 0)>
<Color ### rgb=(0.333333, 0.334641, 0)>
<Color ### rgb=(0.222222, 0.390414, 0)>
<Color ### rgb=(0.111111, 0.446187, 0)>
<Color ### rgb=(0, 0.501961, 0)>
```

In a color-capable terminal, the "###" above will appear to fade between the two specified colors.

Methods are also provided to compare colors for similarity. The simplest algorithm (and the default) is "euclid" which calculates the difference as the distance between them by treating the r, g, b components as coordinates in a 3-dimensional space. The same color will have a distance of 0.0, whilst the largest possible difference is $\sqrt{3}$ (≈ 1.732):

```
>>> c1 = Color('red')
>>> c2 = Color('green')
>>> c3 = c1 * Lightness(0.9)
>>> c1.difference(c2, 'euclid')
1.1189122525867927
>>> c1.difference(c2)
1.1189122525867927
>>> c1.difference(c3)
0.09999999999999998
```

Various Delta-E[7] algorithms (CIE1976, CIE1994, and CIEDE2000) are also provided. In these systems, 2.3 is considered a "just noticeable difference":

```
>>> c1.difference(c2, 'cie1976')
133.10729836196307
>>> c1.difference(c3, 'cie1976')
9.60280542204272
>>> c1.difference(c2, 'cie1994g')
50.97596644678241
>>> c1.difference(c3, 'cie1994g')
5.484832836355026
>>> c1.difference(c2, 'ciede2000')
72.18229138962074
>>> c1.difference(c3, 'ciede2000')
5.490813507834904
```

These algorithms are also available as straight-forward functions:

```
>>> cie1976(c1, c2)
133.10729836196307
>>> ciede2000(c1, c3)
5.490813507834904
```

[7] https://en.wikipedia.org/wiki/Color_difference

CHAPTER

THREE

API - COLOR

The colorzero library includes a comprehensive *Color* (page 7) class which is capable of converting between numerous color representations and calculating color differences. Various ancillary classes can be used to manipulate aspects of a color.

3.1 Color Class

This the primary class in the package, and often the only class you'll need or want to interact with. It has an extremely flexible constructor, along with numerous explicit constructors, and attributes for conversion to other color systems.

class colorzero.**Color**(**args*, ***kwargs*)

The Color class is a tuple which represents a color as linear red, green, and blue components.

The class has a flexible constructor which allows you to create an instance from any built-in color system. There are also explicit constructors for every known system that can convert (directly or indirectly) to linear RGB. For example, an instance of *Color* (page 7) can be constructed in any of the following ways:

```
>>> Color('#f00')
<Color html='#ff0000' rgb=(1, 0, 0)>
>>> Color('green')
<Color html='#008000' rgb=(0.0, 0.501961, 0.0)>
>>> Color(0, 0, 1)
<Color html='#0000ff' rgb=(0, 0, 1)>
>>> Color(h=0, s=1, v=0.5)
<Color html='#800000' rgb=(0.5, 0, 0)>
>>> Color(y=0.4, u=-0.05, v=0.615)
<Color html='#ff104c' rgb=(1, 0.0626644, 0.298394)>
```

The specific forms that the default constructor will accept are enumerated below:

Style	Description
Single scalar parameter	Equivalent to calling *Color.from_string()* (page 11), or *Color.from_rgb24()* (page 11).
Three positional parameters or a 3-tuple with no field names	Equivalent to calling *Color.from_rgb()* (page 11) if all three parameters are between 0.0 and 1.0, or *Color.from_rgb_bytes()* (page 11) otherwise.
Three named parameters, or a 3-tuple with fields “r”, “g”, “b”	
Three named parameters, or a 3-tuple with fields “red”, “green”, “blue”	
Three named parameters, or a 3-tuple with fields “y”, “u”, “v”	Equivalent to calling *Color.from_yuv()* (page 11) if “y” is between 0.0 and 1.0, “u” is between -0.436 and 0.436, and “v” is between -0.615 and 0.615, or *Color.from_yuv_bytes()* (page 11) otherwise.
Three named parameters, or a 3-tuple with fields “y”, “i”, “q”	Equivalent to calling *Color.from_yiq()* (page 11).
Three named parameters, or a 3-tuple with fields “h”, “l”, “s”	Equivalent to calling *Color.from_hls()* (page 11).
Three named parameters, or a 3-tuple with fields “hue”, “lightness”, “saturation”	
Three named parameters, or a 3-tuple with fields “h”, “s”, “v”	Equivalent to calling *Color.from_hsv()* (page 11)
Three named parameters, or a 3-tuple with fields “hue”, “saturation”, “value”	
Three named parameters, or a 3-tuple with fields “x”, “y”, “z”	Equivalent to calling *Color.from_xyz()* (page 11)
Three named parameters, or a 3-tuple with fields “l”, “a”, “b”	Equivalent to calling *Color.from_lab()* (page 11)
Three named parameters, or a 3-tuple with fields “l”, “u”, “v”	Equivalent to calling *Color.from_luv()* (page 11)

If the constructor parameters do not conform to any of the variants in the table above, a `ValueError`[8] will be raised.

Internally, the color is *always* represented as 3 `float`[9] values corresponding to the red, green, and blue components of the color. These values take a value from 0.0 to 1.0 (least to full intensity). The class provides several attributes which can be used to convert one color system into another:

```
>>> Color('#f00').hls
HLS(h=0.0, l=0.5, s=1.0)
>>> Color.from_string('green').hue
Hue(deg=120.0)
>>> Color.from_rgb_bytes(0, 0, 255).yuv
YUV(y=0.114, u=0.436, v=-0.10001426533523537)
```

As *Color* (page 7) derives from tuple, instances are immutable. While this provides the advantage that they can be used in a `set`[10] or as keys of a `dict`[11], it does mean that colors themselves cannot be *directly* manipulated (e.g. by setting the red component).

However, several auxilliary classes in the module provide the ability to perform simple transformations of colors via operators which produce a new *Color* (page 7) instance. For example, you can add, subtract, and multiply colors directly:

```
>>> Color('red') + Color('blue')
<Color html='#ff00ff' rgb=(1, 0, 1)>
>>> Color('magenta') - Color('red')
<Color html='#0000ff' rgb=(0, 0, 1)>
```

Values are clipped to ensure the resulting color is still valid:

```
>>> Color('#ff00ff') + Color('#ff0000')
<Color html='#ff00ff' rgb=(1, 0, 1)>
```

You can wrap numbers in constructors like *Red* (page 15) (or obtain elements of existing colors), then add, subtract, or multiply them with a *Color* (page 7):

```
>>> Color('red') - Red(0.5)
<Color html='#800000' rgb=(0.5, 0, 0)>
>>> Color('green') + Color('grey').red
<Color html='#808000' rgb=(0.501961, 0.501961, 0)>
```

You can even manipulate non-primary attributes like hue, saturation, and lightness with standard addition, subtraction or multiplication operators:

```
>>> Color.from_hls(0.5, 0.5, 1.0)
<Color html='#00ffff' rgb=(0, 1, 1)>
>>> Color.from_hls(0.5, 0.5, 1.0) * Lightness(0.8)
<Color html='#00cccc' rgb=(0, 0.8, 0.8)>
>>> (Color.from_hls(0.5, 0.5, 1.0) * Lightness(0.8)).hls
HLS(h=0.5, l=0.4, s=1.0)
```

In the last example above, a *Color* (page 7) instance is constructed from HLS (hue, lightness, saturation) values with a lightness of 0.5. This is multiplied by a *Lightness* (page 17) a value of 0.8 which constructs a new *Color* (page 7) with the same hue and saturation, but a lightness of 0.4 (0.8×0.5).

If an instance is converted to a string (with `str()`) it will return a string containing the 7-character HTML code for the color (e.g. "#ff0000" for red). As can be seen in the examples above, a similar representation is included for the output of `repr()`[12]. The output of `repr()`[13] can be customized by assigning values to *Color.repr_style* (page 9).

red
Return the red value as a *Red* (page 15) instance

green
Return the green value as a *Green* (page 16) instance

blue
Return the blue value as a *Blue* (page 16) instance

repr_style
Specifies the style of output returned when using `repr()`[14] against a *Color* (page 7) instance. This is an attribute of the class, not of instances. For example:

```
>>> Color('#f00')
<Color html='#ff0000' rgb=(1, 0, 0)>
>>> Color.repr_style = 'html'
>>> Color('#f00')
Color('#ff0000')
```

The following values are valid:

- 'default' - The style shown above
- 'term16m' - Similar to the default style, but instead of the HTML style being included, a swatch previewing the color is output. Note that the terminal must support 24-bit color ANSI codes[15] for this to work.
- 'term256' - Similar to 'term16m', but uses the closest color that can be found in the standard 256-color xterm palette. Note that the terminal must support 8-bit color ANSI codes[16] for this to work.
- 'html' - Outputs a valid *Color* (page 7) constructor using the HTML style, e.g. `Color('#ff99bb')`

- 'rgb' - Outputs a valid *Color* (page 7) constructor using the floating point RGB values, e.g. `Color(1, 0.25, 0)`

difference(*other*, *method='euclid'*)

Determines the difference between this color and *other* using the specified *method*.

Parameters

- **other** (*Color* (page 7)) – The color to compare this color to.
- **method** (*str*[17]) – The algorithm to use in the comparison. Valid values are:
  - 'euclid' - This is the default method. Calculate the Euclidian distance[18]. This is by far the fastest method, but also the least accurate in terms of human perception.
  - 'cie1976' - Use the CIE 1976[19] formula for calculating the difference between two colors in CIE Lab space.
  - 'cie1994g' - Use the CIE 1994[20] formula with the "graphic arts" bias for calculating the difference.
  - 'cie1994t' - Use the CIE 1994[21] forumula with the "textiles" bias for calculating the difference.
  - 'ciede2000' - Use the CIEDE 2000[22] formula for calculating the difference.

Returns A *float*[23] indicating how different the two colors are. Note that the Euclidian distance will be significantly different to the other calculations; effectively this just measures the distance between the two colors by treating them as coordinates in a three dimensional Euclidian space. All other methods are means of calculating a Delta E[24] value in which 2.3 is considered a just-noticeable difference[25] (JND).

For example:

```
>>> Color('red').difference(Color('red'))
0.0
>>> Color('red').difference(Color('red'), method='cie1976')
0.0
>>> Color('red').difference(Color('#900'))
0.4
>>> Color('red').difference(Color('#900'), method='cie1976')
40.17063087142142
>>> Color('red').difference(Color('#900'), method='ciede2000')
21.078146289272155
>>> Color('red').difference(Color('blue'))
1.4142135623730951
>>> Color('red').difference(Color('blue'), method='cie1976')
176.31403908880046
```

Note: Instead of using this method, you may wish to simply use the various difference functions (*euclid()* (page 17), *cie1976()* (page 17), etc.) directly.

classmethod from_cmy(*c*, *m*, *y*)

Construct a *Color* (page 7) from CMY[26] (cyan, magenta, yellow) floats between 0.0 and 1.0.

Note: This conversion uses the basic subtractive method which is not accurate for color reproduction on print devices. See the Color FAQ[27] for more information.

classmethod from_cmyk(*c*, *m*, *y*, *k*)

Construct a *Color* (page 7) from CMYK[28] (cyan, magenta, yellow, black) floats between 0.0 and 1.0.

Note: This conversion uses the basic subtractive method which is not accurate for color reproduction on print devices. See the Color FAQ[29] for more information.

classmethod from_hls(*h*, *l*, *s*)
Construct a *Color* (page 7) from HLS[30] (hue, lightness, saturation) floats between 0.0 and 1.0.

classmethod from_hsv(*h*, *s*, *v*)
Construct a *Color* (page 7) from HSV[31] (hue, saturation, value) floats between 0.0 and 1.0.

classmethod from_lab(*l*, *a*, *b*)
Construct a *Color* (page 7) from (L*, a*, b*) float values representing a color in the CIE Lab color space[32]. The conversion assumes the sRGB working space with reference white D65.

classmethod from_luv(*l*, *u*, *v*)
Construct a *Color* (page 7) from (L*, u*, v*) float values representing a color in the CIE Luv color space[33]. The conversion assumes the sRGB working space with reference white D65.

classmethod from_rgb(*r*, *g*, *b*)
Construct a *Color* (page 7) from three linear RGB[34] float values between 0.0 and 1.0.

classmethod from_rgb24(*n*)
Construct a *Color* (page 7) from an unsigned 24-bit integer number of the form 0x00BBGGRR.

classmethod from_rgb565(*n*)
Construct a *Color* (page 7) from an unsigned 16-bit integer number in RGB565 format.

classmethod from_rgb_bytes(*r*, *g*, *b*)
Construct a *Color* (page 7) from three RGB[35] byte values between 0 and 255.

classmethod from_string(*s*)
Construct a *Color* (page 7) from a 4 or 7 character CSS-like representation (e.g. "#f00" or "#ff0000" for red), or from one of the named colors (e.g. "green" or "wheat") from the CSS standard[36]. Any other string format will result in a `ValueError`[37].

classmethod from_xyz(*x*, *y*, *z*)
Construct a *Color* (page 7) from (X, Y, Z) float values representing a color in the CIE 1931 color space[38]. The conversion assumes the sRGB working space with reference white D65.

classmethod from_yiq(*y*, *i*, *q*)
Construct a *Color* (page 7) from three Y'IQ[39] float values. Y' can be between 0.0 and 1.0, while I and Q can be between -1.0 and 1.0.

classmethod from_yuv(*y*, *u*, *v*)
Construct a *Color* (page 7) from three Y'UV[40] float values. The Y value may be between 0.0 and 1.0. U may be between -0.436 and 0.436, while V may be between -0.615 and 0.615.

classmethod from_yuv_bytes(*y*, *u*, *v*)
Construct a *Color* (page 7) from three Y'UV[41] byte values between 0 and 255. The U and V values are biased by 128 to prevent negative values as is typical in video applications. The Y value is biased by 16 for the same purpose.

gradient(*other*, *steps=10*, *easing=<function linear>*)
Returns a generator which fades between this color and *other* in the specified number of *steps*.

Parameters

- **other** (*Color* (page 7)) – The color that will end the gradient (with the color the method is called upon starting the gradient)
- **steps** (*int*[42]) – The unqiue number of colors to include in the generated gradient. Defaults to 10 if unspecified.
- **easing** (*callable*) – A function which controls the speed of the progression. If specified, if must be a function which takes a single parameter, the number of *steps*, and

yields a sequence of values between 0.0 (representing the start of the gradient) and 1.0 (representing the end). The default is *linear()* (page 18).

Returns A generator yielding *steps* *Color* (page 7) instances which fade from this color to *other*.

For example:

```
>>> Color.repr_style = 'html'
>>> print('\n'.join(
... repr(c) for c in
... Color('red').gradient(Color('green'))
... ))
Color('#ff0000')
Color('#e30e00')
Color('#c61c00')
Color('#aa2b00')
Color('#8e3900')
Color('#714700')
Color('#555500')
Color('#396400')
Color('#1c7200')
Color('#008000')
```

New in version 1.1.

property cmy
Returns a 3-tuple of (cyan, magenta, yellow) float values (between 0.0 and 1.0).

Note: This conversion uses the basic subtractive method which is not accurate for color reproduction on print devices. See the Color FAQ[43] for more information.

---

property cmyk
Returns a 4-tuple of (cyan, magenta, yellow, black) float values (between 0.0 and 1.0).

Note: This conversion uses the basic subtractive method which is not accurate for color reproduction on print devices. See the Color FAQ[44] for more information.

---

property hls
Returns a 3-tuple of (hue, lightness, saturation) float values (between 0.0 and 1.0).

property hsv
Returns a 3-tuple of (hue, saturation, value) float values (between 0.0 and 1.0).

property html
Returns the color as a string in HTML #RRGGBB format.

property hue
Returns the hue of the color as a *Hue* (page 16) instance which can be used in operations with other *Color* (page 7) instances.

property lab
Returns a 3-tuple of (L*, a*, b*) float values representing the color in the CIE Lab color space[45] with the D65 standard illuminant[46].

property lightness
Returns the lightness of the color as a *Lightness* (page 17) instance which can be used in operations with other *Color* (page 7) instances.

property luma
Returns the luma[47] of the color as a *Luma* (page 17) instance which can be used in operations with other *Color* (page 7) instances.

property luv
Returns a 3-tuple of (L*, u*, v*) float values representing the color in the CIE Luv color space[48] with the D65 standard illuminant[49].

property rgb
Return a simple 3-tuple of (r, g, b) float values in the range 0.0 <= n <= 1.0.

Note: The *Color* (page 7) class can already be treated as such a 3-tuple but for the cases where you want a straight `namedtuple()`[50] this property is available.

---

property rgb565
Returns an unsigned 16-bit integer number representing the color in the RGB565 encoding.

property rgb_bytes
Returns a 3-tuple of (red, green, blue) byte values.

property saturation
Returns the saturation of the color as a *Saturation* (page 16) instance which can be used in operations with other *Color* (page 7) instances.

property xyz
Returns a 3-tuple of (X, Y, Z) float values representing the color in the CIE 1931 color space[51]. The conversion assumes the sRGB working space, with reference white D65.

property yiq
Returns a 3-tuple of (y, i, q) float values; y values can be between 0.0 and 1.0, whilst i and q values can be between -1.0 and 1.0.

property yuv
Returns a 3-tuple of (y, u, v) float values; Y values can be between 0.0 and 1.0, U values are between -0.436 and 0.436, and V values are between -0.615 and 0.615.

property yuv_bytes
Returns a 3-tuple of (y, u, v) byte values. Y values are biased by 16 in the result to prevent negatives. U and V values are biased by 128 for the same purpose.

3.2 Format Strings

Instances of *Color* (page 7) can be used in format strings to output various representations of a color, including HTML sequences and ANSI escape sequences to color terminal output. Format specifications can be used to modify the output to support different terminal types. For example:

```
>>> red = Color('red')
>>> green = Color('green')
>>> blue = Color('#47b')
>>> print(f"{red:html}")
#ff0000
>>> print(repr(f"{red}Red{red:0} Alert!"))
'\\x1b[1;31mRed\\x1b[0m Alert!'
>>> print(repr(f"The grass is {green:16m}greener{green:0}."))
'The grass is \\x1b[38;2;0;128;0mgreener\\x1b[0m.'
>>> print(repr(f"{blue:b16m}Blue skies{blue:0}"))
'\\x1b[48;2;68;119;187mBlue skies\\x1b[0m'
```

The format specification is one of:

- "html" - the color will be output as the common 7-character HTML represention of #RRGGBB where RR, GG, and BB are the red, green and blue components expressed as a single hexidecimal byte

[8] https://docs.python.org/3.9/library/exceptions.html#ValueError
[9] https://docs.python.org/3.9/library/functions.html#float
[10] https://docs.python.org/3.9/library/stdtypes.html#set
[11] https://docs.python.org/3.9/library/stdtypes.html#dict
[12] https://docs.python.org/3.9/library/functions.html#repr
[13] https://docs.python.org/3.9/library/functions.html#repr
[14] https://docs.python.org/3.9/library/functions.html#repr
[15] https://en.wikipedia.org/wiki/ANSI_escape_code#24-bit
[16] https://en.wikipedia.org/wiki/ANSI_escape_code#8-bit
[17] https://docs.python.org/3.9/library/stdtypes.html#str
[18] https://en.wikipedia.org/wiki/Euclidean_distance
[19] https://en.wikipedia.org/wiki/Color_difference#CIE76
[20] https://en.wikipedia.org/wiki/Color_difference#CIE94
[21] https://en.wikipedia.org/wiki/Color_difference#CIE94
[22] https://en.wikipedia.org/wiki/Color_difference#CIEDE2000
[23] https://docs.python.org/3.9/library/functions.html#float
[24] https://en.wikipedia.org/wiki/Color_difference
[25] https://en.wikipedia.org/wiki/Just-noticeable_difference
[26] https://en.wikipedia.org/wiki/CMYK_color_model
[27] http://poynton.ca/notes/colour_and_gamma/ColorFAQ.html#RTFToC24
[28] https://en.wikipedia.org/wiki/CMYK_color_model
[29] http://poynton.ca/notes/colour_and_gamma/ColorFAQ.html#RTFToC24
[30] https://en.wikipedia.org/wiki/HSL_and_HSV
[31] https://en.wikipedia.org/wiki/HSL_and_HSV
[32] https://en.wikipedia.org/wiki/Lab_color_space
[33] https://en.wikipedia.org/wiki/CIELUV
[34] https://en.wikipedia.org/wiki/RGB_color_space
[35] https://en.wikipedia.org/wiki/RGB_color_space
[36] http://www.w3.org/TR/css3-color/#svg-color
[37] https://docs.python.org/3.9/library/exceptions.html#ValueError
[38] https://en.wikipedia.org/wiki/CIE_1931_color_space
[39] https://en.wikipedia.org/wiki/YIQ
[40] https://en.wikipedia.org/wiki/YUV
[41] https://en.wikipedia.org/wiki/YUV
[42] https://docs.python.org/3.9/library/functions.html#int
[43] http://poynton.ca/notes/colour_and_gamma/ColorFAQ.html#RTFToC24
[44] http://poynton.ca/notes/colour_and_gamma/ColorFAQ.html#RTFToC24
[45] https://en.wikipedia.org/wiki/Lab_color_space
[46] https://en.wikipedia.org/wiki/Illuminant_D65
[47] https://en.wikipedia.org/wiki/Luma_(video)
[48] https://en.wikipedia.org/wiki/CIELUV
[49] https://en.wikipedia.org/wiki/Illuminant_D65
[50] https://docs.python.org/3.9/library/collections.html#collections.namedtuple
[51] https://en.wikipedia.org/wiki/CIE_1931_color_space

- "css" or "cssrgb" - the color will be output in CSS' functional notation rgb(*r*, *g*, *b*) where *r*, *g*, and *b* are decimal representations of the red, green, and blue components in the range 0 to 255
- "csshsl" - the color will be output in CSS' function notation hue(*h*deg, *s*%, *l*%) where *h*, *s*, and *l* are the hue (expressed in degrees), saturation, and lightness (expressed as percentages)
- One of the ANSI format specifications which consist of an optional foreground / background specifier (the letters "f" or "b") followed by an optional terminal type identifer, which is one of:
  - "8" - the default, indicating only the original 8 DOS colors (black, red, green, yellow, blue, magenta, cyan, and white) are supported. Technically, 16 foreground colors are supported via use of the "bold" style for "intense" colors, if the terminal supports this.
  - "256" - indicates the terminal supports 256 colors via 8-bit color ANSI codes[52]
  - "16m" - indicating the terminal supports ~16 million colors via 24-bit color ANSI codes[53]

"0" can also be specified to indicate that the style should be reset, but this is deprecated. If specified with the optional foreground / background specifier, "0" resets only the foreground / background color. If specified alone it resets all styles. More formally:

```
<term_fore_back> ::= "" | "f" | "b"
<term_type>      ::= "" | "0" | "8" | "256" | "16m"
<term>           ::= <term_fore_back> <term_type>
<html>           ::= "html"
<css>            ::= "css" ("rgb" | "hsl")?
<format_spec>    ::= <html> | <css> | <term>
```

New in version 1.1: The ability to output ANSI codes via format strings, and the customization of `repr()`[54] output.

New in version 1.2: The ability to output HTML and CSS representations via format strings

Deprecated since version 2.1: Use of "0" as a reset indicator; use the new *Default* (page 15) singleton instead

3.3 Default Singleton

The *Default* (page 15) singleton exists as a color which represents the "default" for whatever environment it's rendered in. For example, when using in a format string for CSS, it renders as "inherit" (which is the CSS keyword indicating that a block should inherit its color from its enclosing parent, which is the default). Alternatively, when used with the terminal format strings ("8", "256", "16m") it outputs the ANSI sequence to reset colors to the terminal's default (whatever that may be).

colorzero.**Default = <Color Default>**

The Default singleton is a special value representing the default color for whatever context it is used within. Typically this is only useful in combination with a *Style* (page 19).

3.4 Manipulation Classes

These manipulation classes are used in conjunction with the standard arithmetic addition, subtraction, and multiplication operators to calculate new *Color* (page 7) instances.

class colorzero.**Red**(*x=0*, */*)

Represents the red component of a *Color* (page 7) for use in transformations. Instances of this class can be constructed directly with a float value, or by querying the *Color.red* (page 9) attribute. Addition, subtraction, and multiplication are supported with *Color* (page 7) instances. For example:

[52] https://en.wikipedia.org/wiki/ANSI_escape_code#8-bit
[53] https://en.wikipedia.org/wiki/ANSI_escape_code#24-bit
[54] https://docs.python.org/3.9/library/functions.html#repr

```
>>> Color.from_rgb(0, 0, 0) + Red(0.5)
<Color html='#800000' rgb=(0.5, 0, 0)>
>>> Color('#f00') - Color('#900').red
<Color html='#660000' rgb=(0.4, 0, 0)>
>>> (Red(0.1) * Color('red')).red
Red(0.1)
```

class colorzero.**Green**(*x=0*, */*)

Represents the green component of a *Color* (page 7) for use in transformations. Instances of this class can be constructed directly with a float value, or by querying the *Color.green* (page 9) attribute. Addition, subtraction, and multiplication are supported with *Color* (page 7) instances. For example:

```
>>> Color(0, 0, 0) + Green(0.1)
<Color html='#001a00' rgb=(0, 0.1, 0)>
>>> Color.from_yuv(1, -0.4, -0.6) - Green(1)
<Color html='#510030' rgb=(0.316098, 0, 0.187156)>
>>> (Green(0.5) * Color('white')).rgb
RGB(r=1.0, g=0.5, b=1.0)
```

class colorzero.**Blue**(*x=0*, */*)

Represents the blue component of a *Color* (page 7) for use in transformations. Instances of this class can be constructed directly with a float value, or by querying the *Color.blue* (page 9) attribute. Addition, subtraction, and multiplication are supported with *Color* (page 7) instances. For example:

```
>>> Color(0, 0, 0) + Blue(0.2)
<Color html='#000033' rgb=(0, 0, 0.2)>
>>> Color.from_hls(0.5, 0.5, 1.0) - Blue(1)
<Color html='#00ff00' rgb=(0, 1, 0)>
>>> Blue(0.9) * Color('white')
<Color html='#fffffe6' rgb=(1, 1, 0.9)>
```

class colorzero.**Hue**(*n=None*, *deg=None*, *rad=None*)

Represents the hue of a *Color* (page 7) for use in transformations. Instances of this class can be constructed directly with a float value in the range [0.0, 1.0) representing an angle around the HSL hue wheel[55]. As this is a circular mapping, 0.0 and 1.0 effectively mean the same thing, i.e. out of range values will be normalized into the range [0.0, 1.0).

The class can also be constructed with the keyword arguments `deg` or `rad` if you wish to specify the hue value in degrees or radians instead, respectively. Instances can also be constructed by querying the *Color.hue* (page 12) attribute.

Addition, subtraction, and multiplication are supported with *Color* (page 7) instances. For example:

```
>>> Color(1, 0, 0).hls
HLS(h=0.0, l=0.5, s=1.0)
>>> (Color(1, 0, 0) + Hue(deg=180)).hls
HLS(h=0.5, l=0.5, s=1.0)
```

Note that whilst multiplication by a *Hue* (page 16) doesn't make much sense, it is still supported. However, the circular nature of a hue value can lead to suprising effects. In particular, since 1.0 is equivalent to 0.0 the following may be observed:

```
>>> (Hue(1.0) * Color.from_hls(0.5, 0.5, 1.0)).hls
HLS(h=0.0, l=0.5, s=1.0)
```

property deg

Returns the hue as a value in degrees with the range 0.0 <= n < 360.0.

property rad

Returns the hue as a value in radians with the range 0.0 <= n < 2π.

[55] https://en.wikipedia.org/wiki/Hue

class colorzero.**Saturation**(*x=0*, */*)

Represents the saturation of a *Color* (page 7) for use in transformations. Instances of this class can be constructed directly with a float value, or by querying the *Color.saturation* (page 13) attribute. Addition, subtraction, and multiplication are supported with *Color* (page 7) instances. For example:

```
>>> Color(0.9, 0.9, 0.6) + Saturation(0.1)
<Color html='#ecec93' rgb=(0.925, 0.925, 0.575)>
>>> Color('red') - Saturation(1)
<Color html='#808080' rgb=(0.5, 0.5, 0.5)>
>>> Saturation(0.5) * Color('wheat')
<Color html='#e4d9c3' rgb=(0.896078, 0.85098, 0.766667)>
```

class colorzero.**Lightness**(*x=0*, */*)

Represents the lightness of a *Color* (page 7) for use in transformations. Instances of this class can be constructed directly with a float value, or by querying the *Color.lightness* (page 12) attribute. Addition, subtraction, and multiplication are supported with *Color* (page 7) instances. For example:

```
>>> Color(0, 0, 0) + Lightness(0.1)
<Color html='#1a1a1a' rgb=(0.1, 0.1, 0.1)>
>>> Color.from_rgb_bytes(0x80, 0x80, 0) - Lightness(0.2)
<Color html='#1a1a00' rgb=(0.101961, 0.101961, 0)>
>>> Lightness(0.9) * Color('wheat')
<Color html='#f0ce8e' rgb=(0.94145, 0.806785, 0.555021)>
```

class colorzero.**Luma**(*x=0*, */*)

Represents the luma of a *Color* (page 7) for use in transformations. Instances of this class can be constructed directly with a float value, or by querying the `Color.yuv.y` attribute. Addition, subtraction, and multiplication are supported with *Color* (page 7) instances. For example:

```
>>> Color(0, 0, 0) + Luma(0.1)
<Color html='#1a1a1a' rgb=(0.1, 0.1, 0.1)>
>>> Color('red') * Luma(0.5)
<Color html='#d90000' rgb=(0.8505, 0, 0)>
```

3.5 Difference Functions

colorzero.**euclid**(*color1*, *color2*)

Calculates color difference as a simple Euclidean distance[56] by treating the three components as spatial dimensions.

Note: This function will return considerably different values to the other difference functions. In particular, the maximum "difference" will be $\sqrt{3}$ which is much smaller than the output of the CIE functions.

---

colorzero.**cie1976**(*color1*, *color2*)

Calculates color difference according to the CIE 1976[57] formula. Effectively this is the Euclidean formula, but with CIE L*a*b* components instead of RGB.

colorzero.**cie1994g**(*color1*, *color2*)

Calculates color difference according to the CIE 1994[58] formula with the "textile" bias. See `cie1994()` for further information.

colorzero.**cie1994t**(*color1*, *color2*)

Calculates color difference according to the CIE 1994[59] formula with the "graphics" bias. See `cie1994()` for further information.

[56] https://en.wikipedia.org/wiki/Euclidean_distance
[57] https://en.wikipedia.org/wiki/Color_difference#CIE76
[58] https://en.wikipedia.org/wiki/Color_difference#CIE94

colorzero.**ciede2000**(*color1*, *color2*)
Calculates color difference according to the CIEDE 2000[60] formula. This is the most accurate algorithm currently implemented but also the most complex and slowest. Like CIE1994 it is largely based in CIE L*C*h* space, but with several modifications to account for perceptual uniformity flaws.

3.6 Easing Functions

These functions can be used with the *Color.gradient()* (page 11) method to control the progression of the fade between the two colors.

colorzero.**linear**(*steps*)
Linear easing function; yields *steps* values between 0.0 and 1.0

colorzero.**ease_in**(*steps*)
Quadratic ease-in function; yields *steps* values between 0.0 and 1.0

colorzero.**ease_out**(*steps*)
Quadratic ease-out function; yields *steps* values between 0.0 and 1.0

colorzero.**ease_in_out**(*steps*)
Quadratic ease-in-out function; yields *steps* values between 0.0 and 1.0

[59] https://en.wikipedia.org/wiki/Color_difference#CIE94
[60] https://en.wikipedia.org/wiki/Color_difference#CIEDE2000

CHAPTER

FOUR

API - STYLES

The colorzero library also includes a series of classes which act a bit like stylesheets for formatting large strings. Different classes are used to produce different types of output, such as *HTMLStyles* (page 21) for HTML (and CSS) output, or *TermStyles* (page 21) for ANSI terminal output.

4.1 Style Class

class colorzero.**Style**(*fg*, *bg*=<*Color Default*>)

Represents a named "style" with a foreground (fg) and background (bg) *Color* (page 7) (or *Default* (page 15)).

If *fg* is an instance of *Color* (page 7) it is accepted verbatim. Otherwise, a *Color* (page 7) will be constructed with its value. Likewise for *bg*, which defaults to *Default* (page 15) if not specified.

4.2 Stylesheets

Stylesheets are constructed with an initial mapping of names to suitable style values, or can be constructed with a set of keyword arguments which will be used to create the initial mapping:

```
>>> style = TermStyles({
... 'info': Style(Color('blue'), Default),
... 'warn': Style(Color('white'), Color('red')),
... })
```

The style values can either be a *Style* (page 19) instance, or anything that could be used to construct a *Style* (page 19) instance, including a *Color* (page 7) instance, or valid arguments that could be used to construct a *Color* (page 7) instance, or a tuple of two such values (representing the "fg" and "bg" components respectively). The value None[61] can also be specified, which will be converted to a *Style* (page 19) with *Default* (page 15) as both foreground and background:

```
>>> style2 = TermStyles({
... 'info': 'blue',
... 'warn': ('white', 'red'),
... })
>>> style == style2
True
```

Style mappings are mutable (like an ordinary dict[62]), but keys must be strings, and values will be converted in the same manner as during construction:

[61] https://docs.python.org/3.9/library/constants.html#None
[62] https://docs.python.org/3.9/library/stdtypes.html#dict

```
>>> style['error'] = 'red'
>>> style
TermStyles({'info': Style(fg=<Color html='#0000ff' rgb=(0, 0, 1)>,
bg=<Color Default>), 'warn': Style(fg=<Color html='#ffffff'
rgb=(1, 1, 1)>, bg=<Color html='#ff0000' rgb=(1, 0, 0)>), 'error':
Style(fg=<Color html='#ff0000' rgb=(1, 0, 0)>, bg=<Color Default>)})
```

Style mappings can be used with format strings to generate output in a variety of styles. The format spec for each template must be the name of an entry within the mapping. For example:

```
>>> from colorzero import *
>>> s = TermStyles(warn='red', reset=None)
>>> f'{s:warn}Warning{s:reset}: Do not push the button!'
'\x1b[1;31m\x1b[49mWarning\x1b[0m: Do not push the button!'
```

It is important to bear in mind that the formatting behaviour is stateful. In other words, as the styles are substituted into a string, the instance is keeping track of the current style. This permits descendants like *HTMLStyles* (page 21) to determine when tags need closing or for *TermStyles* (page 21) to determine when to reset foreground or background colors:

```
>>> s = HTMLStyles(warn='red', reset=None)
>>> print(f'{s:warn}This is red ')
<span class="warn">This is red
>>> print('and this will still be red ')
and this will still be red
>>> print(f'because the span has not closed yet{s:reset}')
because the span has not closed yet</span>
```

If the current foreground and background style is not *Default* (page 15) once formatting is complete, you may need to call `reset()` before continuing to format the next string to ensure correct behaviour. The `reset()` method will return whatever string would be needed to reset the current style to the default (if any):

```
>>> print(f'{s:warn}This is red')
<span class="warn">This is red
>>> print(s.reset())
</span>
```

4.3 BaseStyles Class

class colorzero.**BaseStyles**(*styles=None*, ***kwargs*)

Represents a mapping of (arbitrary) names to *Style* (page 19) instances. This is an abstract base class; most users will be more interested in the concrete descendants: *HTMLStyles* (page 21) or *TermStyles* (page 21) which can be used with format strings to produce styled output.

reset()

Reset the "current" style to one with *Default* (page 15) foreground and background.

This is useful when one stylesheet is repeatedly used to format strings, and you wish to guarantee that the style is reset before the next formatting operation. For example:

4.4 StripStyles Class

class colorzero.**StripStyles**(*styles=None*, ***kwargs*)
A degenerate stylesheet class that always returns the empty string for all formatting operations, and the `reset()` method.

4.5 HTMLStyles Class

class colorzero.**HTMLStyles**(*styles=None*, ***, *tag='span'*, ***kwargs*)
A stylesheet that outputs HTML elements[63] when formatting strings.

The name of the elements produced is specified by the *tag* argument, which defaults to "span". Style names must be valid CSS identifiers, or escapable to valid CSS identifiers (in practice, this means no spaces, and no empty strings). A `ValueError`[64] will be raised if you attempt to assign such a key.

The *stylesheet()* (page 21) method can be used to output a valid CSS stylesheet to be used with the generated HTML.

stylesheet(*prefix=''*)
A generator method that yields rules of a CSS stylesheet for all defined styles. The *prefix*, if given, specifies anything that should be output before each rule such as any parent CSS selectors[65].

For example:

```
>>> from colorzero import *
>>> styles = HTMLStyles(warn='red', info='blue', reset=None)
>>> print('\n'.join(styles.stylesheet('div.body ')))
div.body span.warn { color: rgb(255, 0, 0); background-color: inherit; }
div.body span.info { color: rgb(0, 0, 255); background-color: inherit; }
div.body span.reset { color: inherit; background-color: inherit; }
```

4.6 TermStyles Class

class colorzero.**TermStyles**(*styles=None*, ***, *term_colors='8'*, ***kwargs*)
A stylesheet that outputs ANSI escape codes[66].

The *term_colors* argument specifies the sorts of codes produced. This can be:

- "8" - the default, indicating only the original 8 DOS colors (black, red, green, yellow, blue, magenta, cyan, and white) are supported. Technically, 16 foreground colors are supported via use of the "bold" style for "intense" colors, if the terminal supports this.
- "256" - indicates the terminal supports 256 colors via 8-bit color ANSI codes[67]
- "16m" - indicating the terminal supports ~16 million colors via 24-bit color ANSI codes[68]

[63] https://developer.mozilla.org/en-US/docs/Glossary/Element
[64] https://docs.python.org/3.9/library/exceptions.html#ValueError
[65] https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Selectors
[66] https://en.wikipedia.org/wiki/ANSI_escape_code
[67] https://en.wikipedia.org/wiki/ANSI_escape_code#8-bit
[68] https://en.wikipedia.org/wiki/ANSI_escape_code#24-bit

CHAPTER

FIVE

DEVELOPMENT

The main GitHub repository for the project can be found at:

https://github.com/waveform80/colorzero

Anyone is more than welcome to open tickets to discuss bugs, new features, or just to ask usage questions (I find this useful for gauging what questions ought to feature in the FAQ, for example).

Even if you don't feel up to hacking on the code, I'd love to hear suggestions from people of what you'd like the API to look like (even if the code itself isn't particularly pythonic, the interface should be)!

5.1 Development installation

If you wish to develop colorzero itself, it is easiest to obtain the source by cloning the GitHub repository and then use the "develop" target of the Makefile which will install the package as a link to the cloned repository allowing in-place development (it also builds a tags file for use with vim/emacs with Exuberant's ctags utility). The following example demonstrates this method within a virtual Python environment:

```
$ sudo apt install build-essential git \
    exuberant-ctags virtualenvwrapper python-virtualenv python3-virtualenv
$ cd
$ mkvirtualenv -p /usr/bin/python3 colorzero
$ workon colorzero
(colorzero) $ git clone https://github.com/waveform80/colorzero.git
(colorzero) $ cd colorzero
(colorzero) $ make develop
```

To pull the latest changes from git into your clone and update your installation:

```
$ workon colorzero
(colorzero) $ cd ~/colorzero
(colorzero) $ git pull
(colorzero) $ make develop
```

To remove your installation, destroy the sandbox and the clone:

```
(colorzero) $ deactivate
$ rmvirtualenv colorzero
$ rm -fr ~/colorzero
```

5.2 Building the docs

If you wish to build the docs, you'll need a few more dependencies. Inkscape is used for conversion of SVGs to other formats, Graphviz is used for rendering certain charts, and TeX Live is required for building PDF output. The following command should install all required dependencies:

```
$ sudo apt install texlive-latex-recommended texlive-latex-extra \
    texlive-fonts-recommended texlive-xetex graphviz inkscape xindy
```

Once these are installed, you can use the "doc" target to build the documentation:

```
$ workon colorzero
(colorzero) $ cd ~/colorzero
(colorzero) $ make doc
```

The HTML output is written to `build/html` while the PDF output goes to `build/latex`.

5.3 Test suite

If you wish to run the colorzero test suite, follow the instructions in *Development installation* (page 23) above and then make the "test" target within the sandbox:

```
$ workon colorzero
(colorzero) $ cd ~/colorzero
(colorzero) $ make test
```

CHAPTER

SIX

CHANGE LOG

6.1 Release 2.0 (2021-03-15)

- Dropped Python 2.x support. Current Python support level is 3.5 and above.
- Added html and css format specifications to the `Color` (page 7) class' string-formatting capabilities.

6.2 Release 1.1 (2018-05-15)

- Added ability to generate ANSI codes with *Format Strings* (page 14).
- Added `Color.gradient()` (page 11) method.
- Exposed the various difference functions in the API (`euclid()` (page 17), `cie1976()` (page 17), etc).
- Various doc fixes and enhancements.

6.3 Release 1.0 (2018-03-10)

1.0 is the first release after breaking the library out of the picamera[69] project. As this is a 1.x release, API stability will be maintained.

[69] https://github.com/waveform80/picamera

CHAPTER

SEVEN

LICENSE

[70] dave@waveform.org.uk

INDEX