SecretColors$_{o}ldDocumentation$

Release 1.2.0

Rohit Suratekar

Aug 17, 2023

Contents:

1 Palette **3**

2 ColorWheel **23**

3 ColorMaps **27**
- 3.1 ColorMapParent . . . 27
- 3.2 ColorMap . . . 29
- 3.3 BrewerMap . . . 31
- 3.4 TableauMap . . . 42

4 Utils **53**

5 Indices and tables **61**

6 Indices and tables **63**

Python Module Index **65**

Index **67**

It is a python library for fantastic color palettes. You can read more about it from its GitHub website. It can be installed via following PyPi distribution

```
pip install SecretColors
```

This document contains full API reference of the library. There are few private functions which are not included here can be accessed directly via source code on GitHub.

CHAPTER 1

Palette

class SecretColors.**Palette**(*name: str = 'ibm'*, *color_mode: str = 'hex'*, ***, *show_warning: bool = False*, *seed: float = None*, *log: SecretColors.helpers.logging.Log = None*, ***kwargs*)

`Palette` is main class of this library. It provides easy access to huge variety of color manipulations. Entire class is based on internal color database which has been developed based on many Design Systems and famous color palettes. Essentially, we made a simple utility which can copy and paste colors from famous color palettes ;)

Currently, this library supports following Palettes which can be provided at the time of generation of Palette Object.

- **ibm** - IBM Color Palette v2 + v1 [*Default*]
- **material** - Google Material Design Color Palettes
- **material-accent** - Accent colors of Google Material Design Color Palettes
- **brewer** - ColorBrewer Color Palette
- **clarity** - VMWare Clarity Palette
- **tableau** - Tableau Color Palette

```
from SecretColors import Palette
p = Palette() # Generates Default color palette i.e. IBM Color Palette
ibm = Palette("ibm") # Generates IBM Palette
ibm.red() # Returns '#fb4b53'
material = Palette("material") # Generates Material Palette
material.red() # Returns '#f44336'
```

You can specify *color_mode* to control color output format. Currently, this library supports following color modes

- **hex** - Hex Format [*Default*]
- **rgb** - RGB Format (values between 0 and 1)
- **rgba** - RGB with Alpha/Transparency (values between 0 and 1)

- **ahex** - Hex with Alpha/Transparency (Appended before hex)
- **hexa** - Hex with Alpha/Transparency (Appended after hex)

```
p1 = Palette() # Default Color mode (hex)
p1.green() # '#24a148'
p2 = Palette(color_mode="hexa")
p2.green() # '#24a148ff'
p3 = Palette(color_mode="ahex")
p3.green() # '#ff24a148'
p4 = Palette(color_mode="rgb")
p4.green() # (0.141, 0.631, 0.282)
p5 = Palette(color_mode="rgba")
p5.green() # '(0.141, 0.282, 0.631, 1)'
```

Note: *matplotlib* can accepts "hex", "rgb" or "hexa"

__init__(*name: str = 'ibm'*, *color_mode: str = 'hex'*, ***, *show_warning: bool = False*, *seed: float = None*, *log: SecretColors.helpers.logging.Log = None*, ***kwargs*)

Initialize the Palette class

Parameters

- **name** (*str*) – Name of the palette (default: ibm)
- **color_mode** (*str*) – Color mode which will define the output format of each color (default: hex)
- **show_warning** (*bool*) – If True, log will be shown. (default: False)
- **seed** – Seed for Numpy random number generator
- **log** – Log Object
- **kwargs** – Other Arguments (useful if you are subclassing)

name

Returns Name of the current palette

Return type str

version

Returns Version of palette used in SecretColors library

Return type str

creator_url

Returns URL citing original creator

Return type str

get_color_dict

Returns dictionary of color names and their respective default shades

get_color_list

Returns list of all colors with their default shade

cycle(*version: int = 1*, *skip_first: int = 0*)

Creates infinite color cycle

Inspiration is taken from : https://tsitsul.in/blog/coloropt/

```
>>> color_cycle = Palette().cycle()
>>> next(color_cycle) # First Color
>>> next(color_cycle) # Next Color
```

This will go infinitely. After few colors it will start repeating. You can get qualitative colors like following

```
>>> my_colors = [next(color_cycle) for x in range(10)] # Ten colors
```

You may use in the for loop. However, be careful. It is infinite cycle. You need to break the loop by yourself.

Danger: This method creates inifinite number of colors. Be careful while using it in the loop

Parameters **skip_first** – number of colors to be skipped from start. Only

works for first 18 colors. :param version: color sequence version

seed
Current random seed (if any)

colors
Returns dictionary of all colorname and their respective `Color` class.

random(*no_of_colors: int = 1*, ***, *shade: float = None*, *alpha: float = None*, *starting_shade: float = None*, *ending_shade: float = None*, *gradient: bool = True*, *avoid: list = None*, *reverse: bool = False*, *force_list: bool = False*, *print_colors: bool = False*, *seed=None*, ***kwargs*)
Generate random color(s) from current palette.

```
>>> p = Palette()
>>> p.random() # Single random color
>>> p.random(no_of_colors=5)  # 5 random colors
>>> p.random(shade=20) # Random color whos shade is 20
>>> p.random(shade=60, no_of_colors=8) # 8 random colors whose shade is 60
>>> p.random(starting_shade=50, ending_shade=80, no_of_colors=4) #Random␣
↪colors whos shades are between 50-80
>>> p.random(no_of_colors=4, gradient=False) # 4 completely random colors␣
↪with random shades
>>> p.random(no_of_colors=40, avoid=["blue"]) # 40 random colors but do not␣
↪use "blue"
>>> p.random(no_of_colors=10, seed=100) # Generate 10 random color with seed␣
↪100
>>> p.random(force_list=True) # List containing single random color
>>> p.random(print_colors=True) # Print color on the console
>>> p.random(no_of_colors=5, reverse=True) # 5 random colors whos shades␣
↪should arange in darker to lighter
```

Parameters

- **no_of_colors** – Number of colors (default: 1)
- **shade** – Shade of the color (default: palette's default). This will be ignored when number of colors are greater than 1 and starting_shade /ending_shade arguments are provided
- **alpha** – Transparency value (beteen 0-1). This will only be considered if palette 'color_mode' supports Alpha channel. This will be applied to all colors.

- **starting_shade** – Starting shade of colors (used when number of colors are more than 1.)
- **ending_shade** – Ending shade of colors (used when number of colors are more than 1.)
- **gradient** – If True, all shades (not colors) will be sorted in ascending order. (default: True)
- **avoid** – List of colors which should not be considered while generating random numbers. (default: white, black)
- **reverse** – If True, shades will be ordered in descending order
- **force_list** – If True, return type will always be list. Else when no of colors is 1, this function will return str/tuple.
- **print_colors** – If True, colors generated will be printed on the console.
- **seed** – Seed for random number generator (will override the global palette seed)
- **kwargs** – Other named arguments

Returns Str/Tuple/list of random colors depending above options

random_balanced(*no_of_colors: int = 1*)

Generates balanced random colors by defining shade. It essentially just predefines the shade to palettes default shade.

Parameters **no_of_colors** – Number of colors

Returns str/tuple/list based on number of colors and global 'color_mode'

random_gradient(*no_of_colors: int = 3*, ***, *shade: float = None*, *alpha: float = 1*, *print_colors: bool = False*, *complementary=True*, ***kwargs*)

Generates random gradient between two colors

```
>>> p = Palette()
>>> p.random_gradient() # Random gradient between two colors
>>> p.random_gradient(no_of_colors=5) # Generates gradient between 2 colors␣
↪and also adds 3 more colors between them
>>> p.random_gradient(complementary=False) # Use totally random colors for␣
↪the gradient
```

Parameters

- **no_of_colors** – Number of in-between colors (default: 3)
- **shade** – Shades of color
- **alpha** – Alpha value between 0-1 (will ne applied to all colors)
- **print_colors** – If True, prints colors on the console
- **complementary** – If True, generates gradient between two complementary colors. (default: True)
- **kwargs** – Other named arguments

Returns List of colors representing gradient

static cmap_from(*matplotlib*, *hex_color_list: list*)

Creates custom cmap from given hex_color list. Use `ColorMap` for more refined control. Color inputs should be HEX format.

Parameters

- **matplotlib** – matplotlib object (https://matplotlib.org/)
- **hex_color_list** – List of colors in Hex format

Returns *LinearSegmentedColormap* segment which can be used with *matplotlib* plots.

get(*color_name: str*, ***, *shade: float = None*, *no_of_colors: int = 1*, *gradient=True*, *alpha: float = None*, *starting_shade: float = None*, *ending_shade: float = None*, *naming: str = 'w3'*, *strict_search: bool = False*)

This is general methode to retrieve the arbitrary color from the palette. Following steps will be taken,

- It will first check color name present in the current color palette.
- If it is not present, it will search color name in all other available color palettes.
- Then it will also look for common spelling variants of the color through SYNONYM constant from *SecretColors.data.constants* (for example, gray and grey). If any such synonym found, it will return that color.
- Finally it will go through standard color names used in CSS (in case of 'w3') and X11 system (in case of 'x11') and return it.

```
>>> p = Palette()  # IBM Palette
>>> p.get("red") # Default IBM Palette red color (#fa4d56)
>>> p.get("amber") # Amber is not present in IBM so it will return from␣
↪Material palette (#ffc107)
>>> p.get("grey") # It is common variant of 'gray' so it will return regular␣
↪gray (#8d8d8d)
>>> p.get("k") # It also recognize Matplotlib's standard single character␣
↪colors. (#000000)
>>> p.get("aquamarine") # This is not present in any available
>>> # palette. So it will look in common name database and if
>>> # available there, it will return (#7fffd4)
```

if *strict_search* option is True, it will ONLY search in the standard color names provided by respective naming system (w3 or x11).

```
>>> p = Palette()
>>> p.get("gray") # Standard IBM palette gray (#8d8d8d)
>>> p.get("gray", strict_search=True) # w3 common color gray (#808080)
>>> p.get("gray",naming="x11", strict_search=True) # common gray from 'x11'␣
↪colors (#bebebe)
```

Be aware, *strict_search* color name is case sensitive. So take a look at exact color name in respective documentations. These standard names have lot of variants, always refer documentation if getting undesirable results. Current database was updated on 27 August 2020.

w3: https://www.w3.org/TR/css-color-3/#svg-color

x11: https://gitlab.freedesktop.org/xorg/app/rgb/raw/master/rgb.txt

```
>>> p = Palette()
>>> p.get("GhostWhite",naming="x11", strict_search=True) # returns #f8f8ff
>>> p.get("ghostwhite", naming="x11", strict_search=True) # Error
```

Default naming system is set to 'w3'.

Color name found in naming system will be converted into its own pallet by padding white and black at the end. This will enable users to use all usual methods like shades, alpha, etc

```
>>> p = Palette()
>>> p.get("GhostWhite")  # returns #f8f8ff
>>> p.get("GhostWhite", shade=80)  # returns #636366
```

Parameters

- **color_name** – Name of the color
- **shade** – Color Shade (between 0-100). Default will be based on the Palette used. If that is not available 50 will be used.
- **no_of_colors** – Number of colors (default: 1)
- **gradient** – If True, if number of colors are greater than 1, then will be arranged in ascending order of their shade value
- **alpha** – Transparency (between 0-1). Only works in selected color_modes : hexa, ahex, rgba, hsla
- **starting_shade** – If number of colors are more than 1, you can use this to define the starting shade of the color. This does not work when number of colors is 1
- **ending_shade** – If number of colors are more than 1, you can use this to define the ending shade of the color. This does not work when number of colors is 1
- **naming** – Naming system (currently supports w3 or x11). [Default: w3). If name is not found in one system, will be searched in another system.
- **strict_search** – If True, name will be searched only in given naming system. Enabling this will return the default color from given system. Palette colors will be ignored.

Returns ColorString / ColorTuple according to the palette 'color_mode'

red(*, *shade: float = None*, *no_of_colors: int = 1*, *gradient=True*, *alpha: float = None*, *starting_shade: float = None*, *ending_shade: float = None*)
The main function for 'red' color

```
>>> p.red() # Prints default red color from the palette
```

Parameters

- **shade** – Color Shade (between 0-100). Default will be based on the Palette used. If that is not available 50 will be used.
- **no_of_colors** – Number of colors (default: 1)
- **gradient** – If True, if number of colors are greater than 1, then will be arranged in ascending order of their shade value
- **alpha** – Transparency (between 0-1). Only works in selected color_modes : hexa, ahex, rgba, hsla
- **starting_shade** – If number of colors are more than 1, you can use this to define the starting shade of the color. This does not work when number of colors is 1
- **ending_shade** – If number of colors are more than 1, you can use this to define the ending shade of the color. This does not work when number of colors is 1

Returns ColorString (special string class)

blue(*, *shade: float = None*, *no_of_colors: int = 1*, *gradient=True*, *alpha: float = None*, *starting_shade: float = None*, *ending_shade: float = None*)
The main function for 'blue' color

```
>>> p.blue() # Prints default blue color from the palette
```

Parameters

- **shade** – Color Shade (between 0-100). Default will be based on the Palette used. If that is not available 50 will be used.
- **no_of_colors** – Number of colors (default: 1)
- **gradient** – If True, if number of colors are greater than 1, then will be arranged in ascending order of their shade value
- **alpha** – Transparency (between 0-1). Only works in selected color_modes : hexa, ahex, rgba, hsla
- **starting_shade** – If number of colors are more than 1, you can use this to define the starting shade of the color. This does not work when number of colors is 1
- **ending_shade** – If number of colors are more than 1, you can use this to define the ending shade of the color. This does not work when number of colors is 1

Returns ColorString (special string class)

green(*, *shade: float = None*, *no_of_colors: int = 1*, *gradient=True*, *alpha: float = None*, *starting_shade: float = None*, *ending_shade: float = None*)
The main function for 'green' color

```
>>> p.green() # Prints default green color from the palette
```

Parameters

- **shade** – Color Shade (between 0-100). Default will be based on the Palette used. If that is not available 50 will be used.
- **no_of_colors** – Number of colors (default: 1)
- **gradient** – If True, if number of colors are greater than 1, then will be arranged in ascending order of their shade value
- **alpha** – Transparency (between 0-1). Only works in selected color_modes : hexa, ahex, rgba, hsla
- **starting_shade** – If number of colors are more than 1, you can use this to define the starting shade of the color. This does not work when number of colors is 1
- **ending_shade** – If number of colors are more than 1, you can use this to define the ending shade of the color. This does not work when number of colors is 1

Returns ColorString (special string class)

magenta(*, *shade: float = None*, *no_of_colors: int = 1*, *gradient=True*, *alpha: float = None*, *starting_shade: float = None*, *ending_shade: float = None*)
The main function for 'magenta' color

```
>>> p.magenta() # Prints default magenta color from the palette
```

Parameters

- **shade** – Color Shade (between 0-100). Default will be based on the Palette used. If that is not available 50 will be used.
- **no_of_colors** – Number of colors (default: 1)
- **gradient** – If True, if number of colors are greater than 1, then will be arranged in ascending order of their shade value
- **alpha** – Transparency (between 0-1). Only works in selected color_modes : hexa, ahex, rgba, hsla
- **starting_shade** – If number of colors are more than 1, you can use this to define the starting shade of the color. This does not work when number of colors is 1
- **ending_shade** – If number of colors are more than 1, you can use this to define the ending shade of the color. This does not work when number of colors is 1

Returns ColorString (special string class)

purple(*, *shade: float = None*, *no_of_colors: int = 1*, *gradient=True*, *alpha: float = None*, *starting_shade: float = None*, *ending_shade: float = None*)

The main function for 'purple' color

```
>>> p.purple() # Prints default purple color from the palette
```

Parameters

- **shade** – Color Shade (between 0-100). Default will be based on the Palette used. If that is not available 50 will be used.
- **no_of_colors** – Number of colors (default: 1)
- **gradient** – If True, if number of colors are greater than 1, then will be arranged in ascending order of their shade value
- **alpha** – Transparency (between 0-1). Only works in selected color_modes : hexa, ahex, rgba, hsla
- **starting_shade** – If number of colors are more than 1, you can use this to define the starting shade of the color. This does not work when number of colors is 1
- **ending_shade** – If number of colors are more than 1, you can use this to define the ending shade of the color. This does not work when number of colors is 1

Returns ColorString (special string class)

cyan(*, *shade: float = None*, *no_of_colors: int = 1*, *gradient=True*, *alpha: float = None*, *starting_shade: float = None*, *ending_shade: float = None*)

The main function for 'cyan' color

```
>>> p.cyan() # Prints default cyan color from the palette
```

Parameters

- **shade** – Color Shade (between 0-100). Default will be based on the Palette used. If that is not available 50 will be used.
- **no_of_colors** – Number of colors (default: 1)
- **gradient** – If True, if number of colors are greater than 1, then will be arranged in ascending order of their shade value

- **alpha** – Transparency (between 0-1). Only works in selected color_modes : hexa, ahex, rgba, hsla
- **starting_shade** – If number of colors are more than 1, you can use this to define the starting shade of the color. This does not work when number of colors is 1
- **ending_shade** – If number of colors are more than 1, you can use this to define the ending shade of the color. This does not work when number of colors is 1

Returns ColorString (special string class)

teal(*, *shade: float = None*, *no_of_colors: int = 1*, *gradient=True*, *alpha: float = None*, *starting_shade: float = None*, *ending_shade: float = None*)
The main function for 'teal' color

```
>>> p.teal() # Prints default teal color from the palette
```

Parameters

- **shade** – Color Shade (between 0-100). Default will be based on the Palette used. If that is not available 50 will be used.
- **no_of_colors** – Number of colors (default: 1)
- **gradient** – If True, if number of colors are greater than 1, then will be arranged in ascending order of their shade value
- **alpha** – Transparency (between 0-1). Only works in selected color_modes : hexa, ahex, rgba, hsla
- **starting_shade** – If number of colors are more than 1, you can use this to define the starting shade of the color. This does not work when number of colors is 1
- **ending_shade** – If number of colors are more than 1, you can use this to define the ending shade of the color. This does not work when number of colors is 1

Returns ColorString (special string class)

gray_cool(*, *shade: float = None*, *no_of_colors: int = 1*, *gradient=True*, *alpha: float = None*, *starting_shade: float = None*, *ending_shade: float = None*)
The main function for 'gray_cool' color

```
>>> p.gray_cool() # Prints default gray_cool color from the palette
```

Parameters

- **shade** – Color Shade (between 0-100). Default will be based on the Palette used. If that is not available 50 will be used.
- **no_of_colors** – Number of colors (default: 1)
- **gradient** – If True, if number of colors are greater than 1, then will be arranged in ascending order of their shade value
- **alpha** – Transparency (between 0-1). Only works in selected color_modes : hexa, ahex, rgba, hsla
- **starting_shade** – If number of colors are more than 1, you can use this to define the starting shade of the color. This does not work when number of colors is 1
- **ending_shade** – If number of colors are more than 1, you can use this to define the ending shade of the color. This does not work when number of colors is 1

Returns ColorString (special string class)

gray_neutral(*, *shade: float = None*, *no_of_colors: int = 1*, *gradient=True*, *alpha: float = None*, *starting_shade: float = None*, *ending_shade: float = None*)
The main function for 'gray_neutral' color

```
>>> p.gray_neutral() # Prints default gray_neutral color from the palette
```

Parameters

- **shade** – Color Shade (between 0-100). Default will be based on the Palette used. If that is not available 50 will be used.
- **no_of_colors** – Number of colors (default: 1)
- **gradient** – If True, if number of colors are greater than 1, then will be arranged in ascending order of their shade value
- **alpha** – Transparency (between 0-1). Only works in selected color_modes : hexa, ahex, rgba, hsla
- **starting_shade** – If number of colors are more than 1, you can use this to define the starting shade of the color. This does not work when number of colors is 1
- **ending_shade** – If number of colors are more than 1, you can use this to define the ending shade of the color. This does not work when number of colors is 1

Returns ColorString (special string class)

gray(*, *shade: float = None*, *no_of_colors: int = 1*, *gradient=True*, *alpha: float = None*, *starting_shade: float = None*, *ending_shade: float = None*)
The main function for 'gray' color

```
>>> p.gray() # Prints default gray color from the palette
```

Parameters

- **shade** – Color Shade (between 0-100). Default will be based on the Palette used. If that is not available 50 will be used.
- **no_of_colors** – Number of colors (default: 1)
- **gradient** – If True, if number of colors are greater than 1, then will be arranged in ascending order of their shade value
- **alpha** – Transparency (between 0-1). Only works in selected color_modes : hexa, ahex, rgba, hsla
- **starting_shade** – If number of colors are more than 1, you can use this to define the starting shade of the color. This does not work when number of colors is 1
- **ending_shade** – If number of colors are more than 1, you can use this to define the ending shade of the color. This does not work when number of colors is 1

Returns ColorString (special string class)

gray_warm(*, *shade: float = None*, *no_of_colors: int = 1*, *gradient=True*, *alpha: float = None*, *starting_shade: float = None*, *ending_shade: float = None*)
The main function for 'gray_warm' color

```
>>> p.gray_warm() # Prints default gray_warm color from the palette
```

Parameters

- **shade** – Color Shade (between 0-100). Default will be based on the Palette used. If that is not available 50 will be used.
- **no_of_colors** – Number of colors (default: 1)
- **gradient** – If True, if number of colors are greater than 1, then will be arranged in ascending order of their shade value
- **alpha** – Transparency (between 0-1). Only works in selected color_modes : hexa, ahex, rgba, hsla
- **starting_shade** – If number of colors are more than 1, you can use this to define the starting shade of the color. This does not work when number of colors is 1
- **ending_shade** – If number of colors are more than 1, you can use this to define the ending shade of the color. This does not work when number of colors is 1

Returns ColorString (special string class)

red_orange(*, *shade: float = None*, *no_of_colors: int = 1*, *gradient=True*, *alpha: float = None*, *starting_shade: float = None*, *ending_shade: float = None*)

The main function for 'red_orange' color

```
>>> p.red_orange() # Prints default red_orange color from the palette
```

Parameters

- **shade** – Color Shade (between 0-100). Default will be based on the Palette used. If that is not available 50 will be used.
- **no_of_colors** – Number of colors (default: 1)
- **gradient** – If True, if number of colors are greater than 1, then will be arranged in ascending order of their shade value
- **alpha** – Transparency (between 0-1). Only works in selected color_modes : hexa, ahex, rgba, hsla
- **starting_shade** – If number of colors are more than 1, you can use this to define the starting shade of the color. This does not work when number of colors is 1
- **ending_shade** – If number of colors are more than 1, you can use this to define the ending shade of the color. This does not work when number of colors is 1

Returns ColorString (special string class)

black(*, *shade: float = None*, *no_of_colors: int = 1*, *gradient=True*, *alpha: float = None*, *starting_shade: float = None*, *ending_shade: float = None*)

The main function for 'black' color

```
>>> p.black() # Prints default black color from the palette
```

Parameters

- **shade** – Color Shade (between 0-100). Default will be based on the Palette used. If that is not available 50 will be used.

- **no_of_colors** – Number of colors (default: 1)
- **gradient** – If True, if number of colors are greater than 1, then will be arranged in ascending order of their shade value
- **alpha** – Transparency (between 0-1). Only works in selected color_modes : hexa, ahex, rgba, hsla
- **starting_shade** – If number of colors are more than 1, you can use this to define the starting shade of the color. This does not work when number of colors is 1
- **ending_shade** – If number of colors are more than 1, you can use this to define the ending shade of the color. This does not work when number of colors is 1

Returns ColorString (special string class)

white(*, *shade: float = None*, *no_of_colors: int = 1*, *gradient=True*, *alpha: float = None*, *starting_shade: float = None*, *ending_shade: float = None*)

The main function for 'white' color

```
>>> p.white() # Prints default white color from the palette
```

Parameters

- **shade** – Color Shade (between 0-100). Default will be based on the Palette used. If that is not available 50 will be used.
- **no_of_colors** – Number of colors (default: 1)
- **gradient** – If True, if number of colors are greater than 1, then will be arranged in ascending order of their shade value
- **alpha** – Transparency (between 0-1). Only works in selected color_modes : hexa, ahex, rgba, hsla
- **starting_shade** – If number of colors are more than 1, you can use this to define the starting shade of the color. This does not work when number of colors is 1
- **ending_shade** – If number of colors are more than 1, you can use this to define the ending shade of the color. This does not work when number of colors is 1

Returns ColorString (special string class)

ultramarine(*, *shade: float = None*, *no_of_colors: int = 1*, *gradient=True*, *alpha: float = None*, *starting_shade: float = None*, *ending_shade: float = None*)

The main function for 'ultramarine' color

```
>>> p.ultramarine() # Prints default ultramarine color from the palette
```

Parameters

- **shade** – Color Shade (between 0-100). Default will be based on the Palette used. If that is not available 50 will be used.
- **no_of_colors** – Number of colors (default: 1)
- **gradient** – If True, if number of colors are greater than 1, then will be arranged in ascending order of their shade value
- **alpha** – Transparency (between 0-1). Only works in selected color_modes : hexa, ahex, rgba, hsla

- **starting_shade** – If number of colors are more than 1, you can use this to define the starting shade of the color. This does not work when number of colors is 1
- **ending_shade** – If number of colors are more than 1, you can use this to define the ending shade of the color. This does not work when number of colors is 1

Returns ColorString (special string class)

cerulean(*, *shade: float = None*, *no_of_colors: int = 1*, *gradient=True*, *alpha: float = None*, *starting_shade: float = None*, *ending_shade: float = None*)

The main function for 'cerulean' color

```
>>> p.cerulean() # Prints default cerulean color from the palette
```

Parameters

- **shade** – Color Shade (between 0-100). Default will be based on the Palette used. If that is not available 50 will be used.
- **no_of_colors** – Number of colors (default: 1)
- **gradient** – If True, if number of colors are greater than 1, then will be arranged in ascending order of their shade value
- **alpha** – Transparency (between 0-1). Only works in selected color_modes : hexa, ahex, rgba, hsla
- **starting_shade** – If number of colors are more than 1, you can use this to define the starting shade of the color. This does not work when number of colors is 1
- **ending_shade** – If number of colors are more than 1, you can use this to define the ending shade of the color. This does not work when number of colors is 1

Returns ColorString (special string class)

aqua(*, *shade: float = None*, *no_of_colors: int = 1*, *gradient=True*, *alpha: float = None*, *starting_shade: float = None*, *ending_shade: float = None*)

The main function for 'aqua' color

```
>>> p.aqua() # Prints default aqua color from the palette
```

Parameters

- **shade** – Color Shade (between 0-100). Default will be based on the Palette used. If that is not available 50 will be used.
- **no_of_colors** – Number of colors (default: 1)
- **gradient** – If True, if number of colors are greater than 1, then will be arranged in ascending order of their shade value
- **alpha** – Transparency (between 0-1). Only works in selected color_modes : hexa, ahex, rgba, hsla
- **starting_shade** – If number of colors are more than 1, you can use this to define the starting shade of the color. This does not work when number of colors is 1
- **ending_shade** – If number of colors are more than 1, you can use this to define the ending shade of the color. This does not work when number of colors is 1

Returns ColorString (special string class)

lime(*, *shade: float = None*, *no_of_colors: int = 1*, *gradient=True*, *alpha: float = None*, *starting_shade: float = None*, *ending_shade: float = None*)
The main function for 'lime' color

```
>>> p.lime() # Prints default lime color from the palette
```

Parameters

- **shade** – Color Shade (between 0-100). Default will be based on the Palette used. If that is not available 50 will be used.
- **no_of_colors** – Number of colors (default: 1)
- **gradient** – If True, if number of colors are greater than 1, then will be arranged in ascending order of their shade value
- **alpha** – Transparency (between 0-1). Only works in selected color_modes : hexa, ahex, rgba, hsla
- **starting_shade** – If number of colors are more than 1, you can use this to define the starting shade of the color. This does not work when number of colors is 1
- **ending_shade** – If number of colors are more than 1, you can use this to define the ending shade of the color. This does not work when number of colors is 1

Returns ColorString (special string class)

yellow(*, *shade: float = None*, *no_of_colors: int = 1*, *gradient=True*, *alpha: float = None*, *starting_shade: float = None*, *ending_shade: float = None*)
The main function for 'yellow' color

```
>>> p.yellow() # Prints default yellow color from the palette
```

Parameters

- **shade** – Color Shade (between 0-100). Default will be based on the Palette used. If that is not available 50 will be used.
- **no_of_colors** – Number of colors (default: 1)
- **gradient** – If True, if number of colors are greater than 1, then will be arranged in ascending order of their shade value
- **alpha** – Transparency (between 0-1). Only works in selected color_modes : hexa, ahex, rgba, hsla
- **starting_shade** – If number of colors are more than 1, you can use this to define the starting shade of the color. This does not work when number of colors is 1
- **ending_shade** – If number of colors are more than 1, you can use this to define the ending shade of the color. This does not work when number of colors is 1

Returns ColorString (special string class)

gold(*, *shade: float = None*, *no_of_colors: int = 1*, *gradient=True*, *alpha: float = None*, *starting_shade: float = None*, *ending_shade: float = None*)
The main function for 'gold' color

```
>>> p.gold() # Prints default gold color from the palette
```

Parameters

- **shade** – Color Shade (between 0-100). Default will be based on the Palette used. If that is not available 50 will be used.
- **no_of_colors** – Number of colors (default: 1)
- **gradient** – If True, if number of colors are greater than 1, then will be arranged in ascending order of their shade value
- **alpha** – Transparency (between 0-1). Only works in selected color_modes : hexa, ahex, rgba, hsla
- **starting_shade** – If number of colors are more than 1, you can use this to define the starting shade of the color. This does not work when number of colors is 1
- **ending_shade** – If number of colors are more than 1, you can use this to define the ending shade of the color. This does not work when number of colors is 1

Returns ColorString (special string class)

orange(*, *shade: float = None*, *no_of_colors: int = 1*, *gradient=True*, *alpha: float = None*, *starting_shade: float = None*, *ending_shade: float = None*)

The main function for 'orange' color

```
>>> p.orange() # Prints default orange color from the palette
```

Parameters

- **shade** – Color Shade (between 0-100). Default will be based on the Palette used. If that is not available 50 will be used.
- **no_of_colors** – Number of colors (default: 1)
- **gradient** – If True, if number of colors are greater than 1, then will be arranged in ascending order of their shade value
- **alpha** – Transparency (between 0-1). Only works in selected color_modes : hexa, ahex, rgba, hsla
- **starting_shade** – If number of colors are more than 1, you can use this to define the starting shade of the color. This does not work when number of colors is 1
- **ending_shade** – If number of colors are more than 1, you can use this to define the ending shade of the color. This does not work when number of colors is 1

Returns ColorString (special string class)

peach(*, *shade: float = None*, *no_of_colors: int = 1*, *gradient=True*, *alpha: float = None*, *starting_shade: float = None*, *ending_shade: float = None*)

The main function for 'peach' color

```
>>> p.peach() # Prints default peach color from the palette
```

Parameters

- **shade** – Color Shade (between 0-100). Default will be based on the Palette used. If that is not available 50 will be used.
- **no_of_colors** – Number of colors (default: 1)
- **gradient** – If True, if number of colors are greater than 1, then will be arranged in ascending order of their shade value

- **alpha** – Transparency (between 0-1). Only works in selected color_modes : hexa, ahex, rgba, hsla
- **starting_shade** – If number of colors are more than 1, you can use this to define the starting shade of the color. This does not work when number of colors is 1
- **ending_shade** – If number of colors are more than 1, you can use this to define the ending shade of the color. This does not work when number of colors is 1

Returns ColorString (special string class)

violet(*, *shade: float = None*, *no_of_colors: int = 1*, *gradient=True*, *alpha: float = None*, *starting_shade: float = None*, *ending_shade: float = None*)

The main function for 'violet' color

```
>>> p.violet() # Prints default violet color from the palette
```

Parameters

- **shade** – Color Shade (between 0-100). Default will be based on the Palette used. If that is not available 50 will be used.
- **no_of_colors** – Number of colors (default: 1)
- **gradient** – If True, if number of colors are greater than 1, then will be arranged in ascending order of their shade value
- **alpha** – Transparency (between 0-1). Only works in selected color_modes : hexa, ahex, rgba, hsla
- **starting_shade** – If number of colors are more than 1, you can use this to define the starting shade of the color. This does not work when number of colors is 1
- **ending_shade** – If number of colors are more than 1, you can use this to define the ending shade of the color. This does not work when number of colors is 1

Returns ColorString (special string class)

indigo(*, *shade: float = None*, *no_of_colors: int = 1*, *gradient=True*, *alpha: float = None*, *starting_shade: float = None*, *ending_shade: float = None*)

The main function for 'indigo' color

```
>>> p.indigo() # Prints default indigo color from the palette
```

Parameters

- **shade** – Color Shade (between 0-100). Default will be based on the Palette used. If that is not available 50 will be used.
- **no_of_colors** – Number of colors (default: 1)
- **gradient** – If True, if number of colors are greater than 1, then will be arranged in ascending order of their shade value
- **alpha** – Transparency (between 0-1). Only works in selected color_modes : hexa, ahex, rgba, hsla
- **starting_shade** – If number of colors are more than 1, you can use this to define the starting shade of the color. This does not work when number of colors is 1
- **ending_shade** – If number of colors are more than 1, you can use this to define the ending shade of the color. This does not work when number of colors is 1

Returns ColorString (special string class)

pink(*, *shade: float = None*, *no_of_colors: int = 1*, *gradient=True*, *alpha: float = None*, *starting_shade: float = None*, *ending_shade: float = None*)

The main function for 'pink' color

```
>>> p.pink() # Prints default pink color from the palette
```

Parameters

- **shade** – Color Shade (between 0-100). Default will be based on the Palette used. If that is not available 50 will be used.
- **no_of_colors** – Number of colors (default: 1)
- **gradient** – If True, if number of colors are greater than 1, then will be arranged in ascending order of their shade value
- **alpha** – Transparency (between 0-1). Only works in selected color_modes : hexa, ahex, rgba, hsla
- **starting_shade** – If number of colors are more than 1, you can use this to define the starting shade of the color. This does not work when number of colors is 1
- **ending_shade** – If number of colors are more than 1, you can use this to define the ending shade of the color. This does not work when number of colors is 1

Returns ColorString (special string class)

purple_deep(*, *shade: float = None*, *no_of_colors: int = 1*, *gradient=True*, *alpha: float = None*, *starting_shade: float = None*, *ending_shade: float = None*)

The main function for 'purple_deep' color

```
>>> p.purple_deep() # Prints default purple_deep color from the palette
```

Parameters

- **shade** – Color Shade (between 0-100). Default will be based on the Palette used. If that is not available 50 will be used.
- **no_of_colors** – Number of colors (default: 1)
- **gradient** – If True, if number of colors are greater than 1, then will be arranged in ascending order of their shade value
- **alpha** – Transparency (between 0-1). Only works in selected color_modes : hexa, ahex, rgba, hsla
- **starting_shade** – If number of colors are more than 1, you can use this to define the starting shade of the color. This does not work when number of colors is 1
- **ending_shade** – If number of colors are more than 1, you can use this to define the ending shade of the color. This does not work when number of colors is 1

Returns ColorString (special string class)

blue_light(*, *shade: float = None*, *no_of_colors: int = 1*, *gradient=True*, *alpha: float = None*, *starting_shade: float = None*, *ending_shade: float = None*)

The main function for 'blue_light' color

```
>>> p.blue_light() # Prints default blue_light color from the palette
```

Parameters

- **shade** – Color Shade (between 0-100). Default will be based on the Palette used. If that is not available 50 will be used.
- **no_of_colors** – Number of colors (default: 1)
- **gradient** – If True, if number of colors are greater than 1, then will be arranged in ascending order of their shade value
- **alpha** – Transparency (between 0-1). Only works in selected color_modes : hexa, ahex, rgba, hsla
- **starting_shade** – If number of colors are more than 1, you can use this to define the starting shade of the color. This does not work when number of colors is 1
- **ending_shade** – If number of colors are more than 1, you can use this to define the ending shade of the color. This does not work when number of colors is 1

Returns ColorString (special string class)

green_light(*, *shade: float = None*, *no_of_colors: int = 1*, *gradient=True*, *alpha: float = None*, *starting_shade: float = None*, *ending_shade: float = None*)

The main function for 'green_light' color

```
>>> p.green_light() # Prints default green_light color from the palette
```

Parameters

- **shade** – Color Shade (between 0-100). Default will be based on the Palette used. If that is not available 50 will be used.
- **no_of_colors** – Number of colors (default: 1)
- **gradient** – If True, if number of colors are greater than 1, then will be arranged in ascending order of their shade value
- **alpha** – Transparency (between 0-1). Only works in selected color_modes : hexa, ahex, rgba, hsla
- **starting_shade** – If number of colors are more than 1, you can use this to define the starting shade of the color. This does not work when number of colors is 1
- **ending_shade** – If number of colors are more than 1, you can use this to define the ending shade of the color. This does not work when number of colors is 1

Returns ColorString (special string class)

amber(*, *shade: float = None*, *no_of_colors: int = 1*, *gradient=True*, *alpha: float = None*, *starting_shade: float = None*, *ending_shade: float = None*)

The main function for 'amber' color

```
>>> p.amber() # Prints default amber color from the palette
```

Parameters

- **shade** – Color Shade (between 0-100). Default will be based on the Palette used. If that is not available 50 will be used.

- **no_of_colors** – Number of colors (default: 1)
- **gradient** – If True, if number of colors are greater than 1, then will be arranged in ascending order of their shade value
- **alpha** – Transparency (between 0-1). Only works in selected color_modes : hexa, ahex, rgba, hsla
- **starting_shade** – If number of colors are more than 1, you can use this to define the starting shade of the color. This does not work when number of colors is 1
- **ending_shade** – If number of colors are more than 1, you can use this to define the ending shade of the color. This does not work when number of colors is 1

Returns ColorString (special string class)

orange_deep(*, *shade: float = None*, *no_of_colors: int = 1*, *gradient=True*, *alpha: float = None*, *starting_shade: float = None*, *ending_shade: float = None*)

The main function for 'orange_deep' color

```
>>> p.orange_deep() # Prints default orange_deep color from the palette
```

Parameters

- **shade** – Color Shade (between 0-100). Default will be based on the Palette used. If that is not available 50 will be used.
- **no_of_colors** – Number of colors (default: 1)
- **gradient** – If True, if number of colors are greater than 1, then will be arranged in ascending order of their shade value
- **alpha** – Transparency (between 0-1). Only works in selected color_modes : hexa, ahex, rgba, hsla
- **starting_shade** – If number of colors are more than 1, you can use this to define the starting shade of the color. This does not work when number of colors is 1
- **ending_shade** – If number of colors are more than 1, you can use this to define the ending shade of the color. This does not work when number of colors is 1

Returns ColorString (special string class)

brown(*, *shade: float = None*, *no_of_colors: int = 1*, *gradient=True*, *alpha: float = None*, *starting_shade: float = None*, *ending_shade: float = None*)

The main function for 'brown' color

```
>>> p.brown() # Prints default brown color from the palette
```

Parameters

- **shade** – Color Shade (between 0-100). Default will be based on the Palette used. If that is not available 50 will be used.
- **no_of_colors** – Number of colors (default: 1)
- **gradient** – If True, if number of colors are greater than 1, then will be arranged in ascending order of their shade value
- **alpha** – Transparency (between 0-1). Only works in selected color_modes : hexa, ahex, rgba, hsla

- **starting_shade** – If number of colors are more than 1, you can use this to define the starting shade of the color. This does not work when number of colors is 1
- **ending_shade** – If number of colors are more than 1, you can use this to define the ending shade of the color. This does not work when number of colors is 1

Returns ColorString (special string class)

gray_blue(*, *shade: float = None*, *no_of_colors: int = 1*, *gradient=True*, *alpha: float = None*, *starting_shade: float = None*, *ending_shade: float = None*)

The main function for 'gray_blue' color

```
>>> p.gray_blue() # Prints default gray_blue color from the palette
```

Parameters

- **shade** – Color Shade (between 0-100). Default will be based on the Palette used. If that is not available 50 will be used.
- **no_of_colors** – Number of colors (default: 1)
- **gradient** – If True, if number of colors are greater than 1, then will be arranged in ascending order of their shade value
- **alpha** – Transparency (between 0-1). Only works in selected color_modes : hexa, ahex, rgba, hsla
- **starting_shade** – If number of colors are more than 1, you can use this to define the starting shade of the color. This does not work when number of colors is 1
- **ending_shade** – If number of colors are more than 1, you can use this to define the ending shade of the color. This does not work when number of colors is 1

Returns ColorString (special string class)

CHAPTER 2

ColorWheel

class SecretColors.**ColorWheel**(*hex_color: str*)

ColorWheel class is more 'scientific' than using *SecretColors.Palette*. This provides very useful and easy color manipulation tools. This class essentially mimics the typical color wheel. You can 'rotate' the wheel in different directions and axis to get appropriate colors. Following code shows the simplest use

```
cw = ColorWheel("#fa4d56")  # Initialize your 'base color'
print(cw.color)  # Prints #fa4d56
cw.rotate_hue(180)  # Rotates Hue by 180 degree
print(cw.color) # Prints #4dfaf1 (which is on the opposite side of  color wheel_)
cw.rotate_hue(-180)  # Rotate Hue by -180 degree
print(cw.color) # Back to #fa4d56
cw.rotate_lightness(10) # Rotates Lightness by 10
print(cw.color) # Prints #fa5b63
cw.rotate_saturation(-50) # Rotate saturation by -50
print(cw.color)  # Prints #ef676e
```

Tip: When your provide negative rotation values, essentially you are rotating wheel anti-clockwise. Essentially, 0-360 degree represents 0-100% of value. When you go above 360 or below 0, it will automatically wrap it around.

Sometimes, rotation might be little confusing while dealing with lightness. Hence we have special methods which are user-friendly

```
cw = ColorWheel("#fa4d56")  # Initialize your 'base color'
cw.make_darker(30)  # Make current color darker by 30%
print(cw.color)  # Prints #df0612, darker shade of #fa4d56
cw.make_lighter(10)  # Make current color lighter by 30%
print(cw.color)  # Prints #f80915, lighter shade of #df0612
```

You can perform infinite amount of manipulations. Only thing you should remember that, ColorWheel will return the 'current' color. So each time you perform manipulation, color will change. However, at anytime if

you want to reset color to the original color (which you used to initialize the ColorWheel) you can simply use *reset()*.

There are many useful methods which you can use to find colors with specific color harmony (like monochromatic, complementary, etc).

__init__(*hex_color: str*)
Initialize ColorWheel with Hex color. This will be your base color on which all further manipulations can be done.

Parameters **hex_color** – hex color

reset()
Resets all adjustments/manipulation to your original color (which you used while creating this class in *SecretColors.ColorWheel.__init__()*)

color
Returns current color (which has all the manipulations)

rotate_hue(*angle: float*)
Rotates Hue with given angle

Parameters **angle** (*float*) – Angle of rotation

rotate_saturation(*angle: float*)
Rotates Saturation with given angle

Parameters **angle** (*float*) – Angle of rotation

rotate_lightness(*angle: float*)
Rotates Lightness with given angle

Parameters **angle** (*float*) – Angle of ratation

make_darker(*percentage: float*)
Makes color darker by reducing lightness

Parameters **percentage** (*float*) – Percentage change

make_lighter(*percentage: float*)
Makes color lighter by increasing lightness

Parameters **percentage** (*float*) – Percentage change

complementary(*is_reversed: bool = False*) → list
Generates two complementary colors. Out of which, one will be your current color

Parameters **is_reversed** (*bool*) – If True, return list will be reversed

Returns List of two complementary colors

triadic(*is_reserved: bool = False*) → list
Generate triadic colors. Out of which, one will be your current color

Parameters **is_reserved** (*bool*) – If True, return list will be reversed

Returns List of 3 triadic colors

tetradic(*is_reversed: bool = False*) → list
Generates tetradic colors. Out of which, one will be your current color.

Parameters **is_reversed** (*bool*) – If True, return list will be reversed

Returns List of 4 tetradic colors

analogous(*loc='first'*, *is_reversed: bool = False*) → list
Generate list of analogous colors. Out of which, one will be your current color. Location of where current color should be in these 3 colors can be decided by *loc* argument.

Parameters

- **loc** (*str*) – Location of current color in the analogous colors. Available options: [first, middle, last]. (default: first)
- **is_reversed** (*bool*) – If True, return list will be reversed.

Returns List of 3 analogous colors

text_color() → str
Simply returns black or white color based on the color contrast of back/white with current color. This will be useful when you are writing text on any colored background.

Returns White of Black color (based on contrast)

CHAPTER 3

ColorMaps

ColorMaps are essential part of any standard data visualization. Since beginning of inception of this library, one of the aim was to provide easy access to diverse colormaps to the users. In this release (v.1.2+), we have added full support to ColorMaps.

What is difference between ColorMap and Palette?

You must have heard of 'color palettes' when you are searching for good colors on the internet. We have entire class *Palette* which is dedicated for generating colors from different standard libraries and color lists. Then question arises, what are these 'ColorMaps' we are referring to?

We used word 'ColorMap' to the object which can provide easy access to the python objects which can be used in various `matplotlib` functions. Matplotlib library uses `cmap` (colormaps) in their library. Which are essentially either `matplotlib.colors.ListedColormap` or `matplotlib.colors.LinearSegmentedColormap` classes. They provide nice access to select color based on its value. You can read more about them on their website .

Hence we have made these ColorMaps which can be directly used in `matplotlib` workflow inplace of in-build colormaps. However, it is not restricted to matplotlib. You can use these object to generate variety of colors based on value. Or you can just get colors from these ColorMaps and use them in regular workflow.

3.1 ColorMapParent

class SecretColors.cmaps.parent.**ColorMapParent**(*matplotlib*, *palette: SecretColors.models.palette.Palette = None*, *log: SecretColors.helpers.logging.Log = None*, *seed=None*)

This is parent class which will be inherited by all ColorMap objects. It includes all basic methods which will be common to all the ColorMaps.

> **Danger:** Do not use this class in your workflow. This class is intended as a parent class which you can inherit to make new colormaps. For general purpose use, you should use *ColorMap* instead.

__init__ (*matplotlib*, *palette: SecretColors.models.palette.Palette = None*, *log: SecretColors.helpers.logging.Log = None*, *seed=None*)
Initializing of any ColorMap.

Parameters

- **matplotlib** – matplotlib object from matplotlib library
- **palette** (*Palette*) – Palette from which you want colors
- **log** (*Log*) – Log class
- **seed** – Seed for random number generation

data
Returns all available ColorMap data. This is valid ONLY for special subclass (e.g. BrewerMap). It will return None for 'ColorMap' class.

Return type dict

palette

Returns Returns current palette from which colors are drawn

Return type *Palette*

get_all
Returns list of available special colormaps. This works only with special subclasses like BrewerMap.

Returns List of colormap names

Return type List[str]

get_colors (*name: str*, *no_of_colors: int*) → list
This is easy way to get the available colors in current colormap

```
cm = BrewerMap(matplotlib)
cm.get_colors('Spectral', 9) # Returns 9 'Spectral' colors from BrewerMap␣
↪colormap
```

Warning: Be careful in using *no_of_colors* argument. It actually points to number of colors available in given colormap. For example, 'Tableau' map from *TableauMap* contains two list of colors, 10 and 20. So you need to enter either 10 or 20. Any other number will raise ValueError. You can check which all options are available by *get_all* property. More about this can be read in documentation of *get()* function.

Parameters

- **name** (*str*) – Name of the special colormap
- **no_of_colors** (*int*) – Number of colors (see warning above)

Returns List of colors

Return type List[str]

Raises ValueError (if used on *ColorMap* or wrong *no_of_colors* provided)

from_list (*color_list: list*, *is_qualitative: bool = False*, *is_reversed=False*)
You can create your own colormap with list of own colors

Parameters

- **color_list** – List of colors
- **is_qualitative** – If True, makes listed colormap
- **is_reversed** – Reverses the order of color in Colormap

Returns Colormap which can be directly used with matplotlib

get (*name: str*, ***, *no_of_colors: int = None*, *is_qualitative: bool = False*, *is_reversed=False*)
Get arbitrary color map from current ColorMap object

no_of_colors is probably the most important parameter in the colormap classes. In this library each colormap data is structured in the form of dictionary as shown below:

```
data = { 'map_name' : {
        '10': [c1, c2, ... c10],
        '5' : [b1, b2, ... b5],
        ...
        'type': Type of colormap
    }
}
```

In above example, if you want to access list [c1, c2...c10], you can do following,

```
>>> YourMap().get('map_name',no_of_colors=10) # Returns [c1, c2 ...c10]
```

You can check which all colormaps are available by *get_all* property

Parameters

- **name** (*str*) – Exact Name of the Colormap
- **no_of_colors** (*int*) – Number of colors. (See discussion above)
- **is_qualitative** (*bool*) – If True, listed colormap will be returned. (default: False)
- **is_reversed** (*bool*) – If True, colormap will be reversed. (default: False)

Returns Colormap object

Return type matplotlib.colors.ListedColormap or matplotlib.colors.LinearSegmentedColormap

3.2 ColorMap

class SecretColors.cmaps.**ColorMap**(*matplotlib*, *palette: SecretColors.models.palette.Palette = None*, *log: SecretColors.helpers.logging.Log = None*, *seed=None*)

This is simple wrapper around *ColorMapParent*. This wrapper let you utilize all methods from its parent class. For all general purpose use, you should use this class. If you want more specialized ColorMaps, use their respective classes. Following is the simplest use where you want to visualize your data in typical 'greens' palette

```
import matplotlib
import matplotlib.pyplot as plt
from SecretColors.cmaps import ColorMap
import numpy as np

cm = ColorMap(matplotlib)
data = np.random.rand(5, 5)
```

(continues on next page)

(continued from previous page)

```
plt.imshow(data, cmap=cm.greens())
plt.colorbar()
plt.show()
```

You can easily change standard colormaps like following

```
cm.reds()  # Reds colormap
cm.oranges() # Oranges colormap
cm.blues()  # Blues colormap
cm.grays()  # Grays colormap
```

All standard colormaps accepts following basic options (which should be provided as a named arguments)

- **no_of_colors** Number of colors you want in your colormap. It usually defines how smaooth your color gradient will be
- **starting_shade** What will be the first shade of your colormap
- **ending_shade** What will be the last shade of your colormap
- **is_qualitative** If True, `matplotlib.colors.ListedColormap` will be used instead `matplotlib.colors.LinearSegmentedColormap`. Essentially it will provide discrete colormap instead linear
- **is_reversed** If True, colormap will be reversed

```
cm.purples(no_of_colors=8)
cm.greens(starting_shade=30, ending_shade=80)
cm.blues(is_qualitative=True)
cm.reds(ending_shade=50, is_reversed=True, no_of_colors=5)
```

You can mix-and-match every argument. Essentially there are infinite possibilities.

If you want even more fine-tune control over your colormap, you can use your own colormaps by `from_list()` method.

```
cm = ColorMap(matplotlib)
p = Palette()
my_colors = [p.red(shade=30), p.white(), p.blue(shade=60)]
my_cmap = cm.from_list(my_colors)
plt.imshow(data, cmap=my_cmap)
```

We have some in-build color lists for divergent colormaps. You can use `random_divergent()` for its easy access. Read *ColorMapParent* documentation for more details on helper functions.

If you like colors from specific *Palette*, you can easily switch all colors with single line

```
cm = ColorMap(matplotlib)
cm.palette = Palette("material") # Material Palette colors will be used.
cm.palette = Palette("brewer") # ColorBrewer colors will be used.
```

Tip: For "brewer" and "tableau", you should prefer using *BrewerMap* and *TableauMap* intsead just changing palette here. As these classes will provide you much more additional methods which are only available in those classes.

---

data
Returns all available ColorMap data. This is valid ONLY for special subclass (e.g. BrewerMap). It will return None for 'ColorMap' class.

Return type dict

3.3 BrewerMap

class SecretColors.cmaps.**BrewerMap**(*matplotlib*)
ColorBrewer2 is probably one of the best known colormap, special in the cartography community. It provides very robust and nice color maps which many people use in their daily visualization needs. We provide all the functionality of ColorBrewer maps in this class. You can take a look at their site for more details.

Simplest way to use this class in your `matplotlib` workflow is following

```
import matplotlib
import matplotlib.pyplot as plt
from SecretColors.cmaps import BrewerMap
bm = BrewerMap(matplotlib)
plt.imshow(data, cmap=bm.spectral())
plt.show()
```

Here, *spectral()* is one of the available standard color map in this class. If you know the exact name of the color map, you can use following as well

```
plt.imshow(data, cmap= bm.get("Spectral")) # Alternate way
bm.get_all  # Gives you all available colormaps
bm.data # Returns available color data
```

To know which all color maps are available in current class you can use `get_all` attribute. These methods can be heavily customized. Take a look at *ColorMapParent* documentation.

List of available colormaps in BrewerMap

- Spectral
- RdYlGn
- RdBu
- PiYG
- PRGn
- RdYlBu
- BrBG
- RdGy
- PuOr
- Set2
- Accent
- Set1
- Set3
- Dark2
- Paired
- Pastel2
- Pastel1
- OrRd
- PuBu
- BuPu

- Oranges
- BuGn
- YlOrBr
- YlGn
- Reds
- RdPu
- Greens
- YlGnBu
- Purples
- GnBu
- Greys
- YlOrRd
- PuRd
- Blues
- PuBuGn

---

data
Returns all available ColorMap data. This is valid ONLY for special subclass (e.g. BrewerMap). It will return None for 'ColorMap' class.

Return type dict

spectral (*, *no_of_colors: int = None*, *is_qualitative: bool = False*, *is_reversed=False*)
This is special method available for *SecretColors.cmaps.BrewerMap* class. This function provides easy access to *Spectral* colormap. You can also use `get()` method to achieve the same result.

```
from SecretColors.cmaps import BrewerMap
import matplotlib
cm = BrewerMap(matplotlib)
plt.imshow(data, cm=cm.spectral()) # Or cm.get('Spectral')
plt.show()
```

Parameters

- **no_of_colors** – No of colors
- **is_qualitative** – If True, ColorMap will be qualitative
- **is_reversed** – If True, ColorMap will be reversed

Returns Matplotlib *LinearSegmentedColormap* or *ListedColormap* based on the options provided. You can use this directly in the matplotlib

rd_yl_gn (*, *no_of_colors: int = None*, *is_qualitative: bool = False*, *is_reversed=False*)
This is special method available for *SecretColors.cmaps.BrewerMap* class. This function provides easy access to *RgYlGn* colormap. You can also use `get()` method to achieve the same result.

```
from SecretColors.cmaps import BrewerMap
import matplotlib
cm = BrewerMap(matplotlib)
plt.imshow(data, cm=cm.rd_yl_gn()) # Or cm.get('RgYlGn')
plt.show()
```

Parameters

- **no_of_colors** – No of colors
- **is_qualitative** – If True, ColorMap will be qualitative

- **is_reversed** – If True, ColorMap will be reversed

Returns Matplotlib *LinearSegmentedColormap* or *ListedColormap* based on the options provided. You can use this directly in the matplotlib

rd_bu (*, *no_of_colors: int = None*, *is_qualitative: bool = False*, *is_reversed=False*)

This is special method available for *SecretColors.cmaps.BrewerMap* class. This function provides easy access to *RdBu* colormap. You can also use `get()` method to achieve the same result.

```
from SecretColors.cmaps import BrewerMap
import matplotlib
cm = BrewerMap(matplotlib)
plt.imshow(data, cm=cm.rd_bu()) # Or cm.get('RdBu')
plt.show()
```

Parameters

- **no_of_colors** – No of colors
- **is_qualitative** – If True, ColorMap will be qualitative
- **is_reversed** – If True, ColorMap will be reversed

Returns Matplotlib *LinearSegmentedColormap* or *ListedColormap* based on the options provided. You can use this directly in the matplotlib

pi_yg (*, *no_of_colors: int = None*, *is_qualitative: bool = False*, *is_reversed=False*)

This is special method available for *SecretColors.cmaps.BrewerMap* class. This function provides easy access to *PiYG* colormap. You can also use `get()` method to achieve the same result.

```
from SecretColors.cmaps import BrewerMap
import matplotlib
cm = BrewerMap(matplotlib)
plt.imshow(data, cm=cm.pi_yg()) # Or cm.get('PiYG')
plt.show()
```

Parameters

- **no_of_colors** – No of colors
- **is_qualitative** – If True, ColorMap will be qualitative
- **is_reversed** – If True, ColorMap will be reversed

Returns Matplotlib *LinearSegmentedColormap* or *ListedColormap* based on the options provided. You can use this directly in the matplotlib

pr_gn (*, *no_of_colors: int = None*, *is_qualitative: bool = False*, *is_reversed=False*)

This is special method available for *SecretColors.cmaps.BrewerMap* class. This function provides easy access to *PRGn* colormap. You can also use `get()` method to achieve the same result.

```
from SecretColors.cmaps import BrewerMap
import matplotlib
cm = BrewerMap(matplotlib)
plt.imshow(data, cm=cm.pr_gn()) # Or cm.get('PRGn')
plt.show()
```

Parameters

- **no_of_colors** – No of colors
- **is_qualitative** – If True, ColorMap will be qualitative
- **is_reversed** – If True, ColorMap will be reversed

Returns Matplotlib *LinearSegmentedColormap* or *ListedColormap* based on the options provided. You can use this directly in the matplotlib

rd_yl_bu(*, *no_of_colors: int = None*, *is_qualitative: bool = False*, *is_reversed=False*)

This is special method available for *SecretColors.cmaps.BrewerMap* class. This function provides easy access to *RdYlBu* colormap. You can also use `get()` method to achieve the same result.

```
from SecretColors.cmaps import BrewerMap
import matplotlib
cm = BrewerMap(matplotlib)
plt.imshow(data, cm=cm.rd_yl_bu()) # Or cm.get('RdYlBu')
plt.show()
```

Parameters

- **no_of_colors** – No of colors
- **is_qualitative** – If True, ColorMap will be qualitative
- **is_reversed** – If True, ColorMap will be reversed

Returns Matplotlib *LinearSegmentedColormap* or *ListedColormap* based on the options provided. You can use this directly in the matplotlib

br_bg(*, *no_of_colors: int = None*, *is_qualitative: bool = False*, *is_reversed=False*)

This is special method available for *SecretColors.cmaps.BrewerMap* class. This function provides easy access to *BrBG* colormap. You can also use `get()` method to achieve the same result.

```
from SecretColors.cmaps import BrewerMap
import matplotlib
cm = BrewerMap(matplotlib)
plt.imshow(data, cm=cm.br_bg()) # Or cm.get('BrBG')
plt.show()
```

Parameters

- **no_of_colors** – No of colors
- **is_qualitative** – If True, ColorMap will be qualitative
- **is_reversed** – If True, ColorMap will be reversed

Returns Matplotlib *LinearSegmentedColormap* or *ListedColormap* based on the options provided. You can use this directly in the matplotlib

rd_gy(*, *no_of_colors: int = None*, *is_qualitative: bool = False*, *is_reversed=False*)

This is special method available for *SecretColors.cmaps.BrewerMap* class. This function provides easy access to *RdGy* colormap. You can also use `get()` method to achieve the same result.

```
from SecretColors.cmaps import BrewerMap
import matplotlib
cm = BrewerMap(matplotlib)
plt.imshow(data, cm=cm.rd_gy()) # Or cm.get('RdGy')
plt.show()
```

Parameters

- **no_of_colors** – No of colors
- **is_qualitative** – If True, ColorMap will be qualitative
- **is_reversed** – If True, ColorMap will be reversed

Returns Matplotlib *LinearSegmentedColormap* or *ListedColormap* based on the options provided. You can use this directly in the matplotlib

pu_or (*, *no_of_colors: int = None*, *is_qualitative: bool = False*, *is_reversed=False*)

This is special method available for *SecretColors.cmaps.BrewerMap* class. This function provides easy access to *PuOr* colormap. You can also use `get()` method to achieve the same result.

```
from SecretColors.cmaps import BrewerMap
import matplotlib
cm = BrewerMap(matplotlib)
plt.imshow(data, cm=cm.pu_or()) # Or cm.get('PuOr')
plt.show()
```

Parameters

- **no_of_colors** – No of colors
- **is_qualitative** – If True, ColorMap will be qualitative
- **is_reversed** – If True, ColorMap will be reversed

Returns Matplotlib *LinearSegmentedColormap* or *ListedColormap* based on the options provided. You can use this directly in the matplotlib

set1 (*, *no_of_colors: int = None*, *is_qualitative: bool = False*, *is_reversed=False*)

This is special method available for *SecretColors.cmaps.BrewerMap* class. This function provides easy access to *Set1* colormap. You can also use `get()` method to achieve the same result.

```
from SecretColors.cmaps import BrewerMap
import matplotlib
cm = BrewerMap(matplotlib)
plt.imshow(data, cm=cm.set1()) # Or cm.get('Set1')
plt.show()
```

Parameters

- **no_of_colors** – No of colors
- **is_qualitative** – If True, ColorMap will be qualitative
- **is_reversed** – If True, ColorMap will be reversed

Returns Matplotlib *LinearSegmentedColormap* or *ListedColormap* based on the options provided. You can use this directly in the matplotlib

set2 (*, *no_of_colors: int = 8*, *is_qualitative: bool = False*, *is_reversed=False*)

This is special method available for *SecretColors.cmaps.BrewerMap* class. This function provides easy access to *Set2* colormap. You can also use `get()` method to achieve the same result.

```
from SecretColors.cmaps import BrewerMap
import matplotlib
cm = BrewerMap(matplotlib)
```

(continues on next page)

(continued from previous page)

```
plt.imshow(data, cm=cm.set2()) # Or cm.get('Set2')
plt.show()
```

Parameters

- **no_of_colors** – No of colors
- **is_qualitative** – If True, ColorMap will be qualitative
- **is_reversed** – If True, ColorMap will be reversed

Returns Matplotlib *LinearSegmentedColormap* or *ListedColormap* based on the options provided. You can use this directly in the matplotlib

accent(*, *no_of_colors: int = 8*, *is_qualitative: bool = False*, *is_reversed=False*)

This is special method available for *SecretColors.cmaps.BrewerMap* class. This function provides easy access to *Accent* colormap. You can also use `get()` method to achieve the same result.

```
from SecretColors.cmaps import BrewerMap
import matplotlib
cm = BrewerMap(matplotlib)
plt.imshow(data, cm=cm.accent()) # Or cm.get('Accent')
plt.show()
```

Parameters

- **no_of_colors** – No of colors
- **is_qualitative** – If True, ColorMap will be qualitative
- **is_reversed** – If True, ColorMap will be reversed

Returns Matplotlib *LinearSegmentedColormap* or *ListedColormap* based on the options provided. You can use this directly in the matplotlib

set3(*, *no_of_colors: int = None*, *is_qualitative: bool = False*, *is_reversed=False*)

This is special method available for *SecretColors.cmaps.BrewerMap* class. This function provides easy access to *Set3* colormap. You can also use `get()` method to achieve the same result.

```
from SecretColors.cmaps import BrewerMap
import matplotlib
cm = BrewerMap(matplotlib)
plt.imshow(data, cm=cm.set3()) # Or cm.get('Set3')
plt.show()
```

Parameters

- **no_of_colors** – No of colors
- **is_qualitative** – If True, ColorMap will be qualitative
- **is_reversed** – If True, ColorMap will be reversed

Returns Matplotlib *LinearSegmentedColormap* or *ListedColormap* based on the options provided. You can use this directly in the matplotlib

dark2(*, *no_of_colors: int = 8*, *is_qualitative: bool = False*, *is_reversed=False*)

This is special method available for *SecretColors.cmaps.BrewerMap* class. This function provides easy access to *Dark2* colormap. You can also use `get()` method to achieve the same result.

```
from SecretColors.cmaps import BrewerMap
import matplotlib
cm = BrewerMap(matplotlib)
plt.imshow(data, cm=cm.dark2()) # Or cm.get('Dark2')
plt.show()
```

Parameters

- **no_of_colors** – No of colors
- **is_qualitative** – If True, ColorMap will be qualitative
- **is_reversed** – If True, ColorMap will be reversed

Returns Matplotlib *LinearSegmentedColormap* or *ListedColormap* based on the options provided. You can use this directly in the matplotlib

paired(*, *no_of_colors: int = None*, *is_qualitative: bool = False*, *is_reversed=False*)

This is special method available for *SecretColors.cmaps.BrewerMap* class. This function provides easy access to *Paired* colormap. You can also use `get()` method to achieve the same result.

```
from SecretColors.cmaps import BrewerMap
import matplotlib
cm = BrewerMap(matplotlib)
plt.imshow(data, cm=cm.paired()) # Or cm.get('Paired')
plt.show()
```

Parameters

- **no_of_colors** – No of colors
- **is_qualitative** – If True, ColorMap will be qualitative
- **is_reversed** – If True, ColorMap will be reversed

Returns Matplotlib *LinearSegmentedColormap* or *ListedColormap* based on the options provided. You can use this directly in the matplotlib

pastel2(*, *no_of_colors: int = 8*, *is_qualitative: bool = False*, *is_reversed=False*)

This is special method available for *SecretColors.cmaps.BrewerMap* class. This function provides easy access to *Pastel2* colormap. You can also use `get()` method to achieve the same result.

```
from SecretColors.cmaps import BrewerMap
import matplotlib
cm = BrewerMap(matplotlib)
plt.imshow(data, cm=cm.pastel2()) # Or cm.get('Pastel2')
plt.show()
```

Parameters

- **no_of_colors** – No of colors
- **is_qualitative** – If True, ColorMap will be qualitative
- **is_reversed** – If True, ColorMap will be reversed

Returns Matplotlib *LinearSegmentedColormap* or *ListedColormap* based on the options provided. You can use this directly in the matplotlib

pastel1(*, *no_of_colors: int = None*, *is_qualitative: bool = False*, *is_reversed=False*)
This is special method available for *SecretColors.cmaps.BrewerMap* class. This function provides easy access to *Pastel1* colormap. You can also use `get()` method to achieve the same result.

```
from SecretColors.cmaps import BrewerMap
import matplotlib
cm = BrewerMap(matplotlib)
plt.imshow(data, cm=cm.pastel1()) # Or cm.get('Pastel1')
plt.show()
```

Parameters

- **no_of_colors** – No of colors
- **is_qualitative** – If True, ColorMap will be qualitative
- **is_reversed** – If True, ColorMap will be reversed

Returns Matplotlib *LinearSegmentedColormap* or *ListedColormap* based on the options provided. You can use this directly in the matplotlib

or_rd(*, *no_of_colors: int = None*, *is_qualitative: bool = False*, *is_reversed=False*)
This is special method available for *SecretColors.cmaps.BrewerMap* class. This function provides easy access to *OrRd* colormap. You can also use `get()` method to achieve the same result.

```
from SecretColors.cmaps import BrewerMap
import matplotlib
cm = BrewerMap(matplotlib)
plt.imshow(data, cm=cm.or_rd()) # Or cm.get('OrRd')
plt.show()
```

Parameters

- **no_of_colors** – No of colors
- **is_qualitative** – If True, ColorMap will be qualitative
- **is_reversed** – If True, ColorMap will be reversed

Returns Matplotlib *LinearSegmentedColormap* or *ListedColormap* based on the options provided. You can use this directly in the matplotlib

pu_bu(*, *no_of_colors: int = None*, *is_qualitative: bool = False*, *is_reversed=False*)
This is special method available for *SecretColors.cmaps.BrewerMap* class. This function provides easy access to *PuBU* colormap. You can also use `get()` method to achieve the same result.

```
from SecretColors.cmaps import BrewerMap
import matplotlib
cm = BrewerMap(matplotlib)
plt.imshow(data, cm=cm.pu_bu()) # Or cm.get('PuBU')
plt.show()
```

Parameters

- **no_of_colors** – No of colors
- **is_qualitative** – If True, ColorMap will be qualitative
- **is_reversed** – If True, ColorMap will be reversed

Returns Matplotlib *LinearSegmentedColormap* or *ListedColormap* based on the options provided. You can use this directly in the matplotlib

bu_pu (*, *no_of_colors: int = None*, *is_qualitative: bool = False*, *is_reversed=False*)

This is special method available for *SecretColors.cmaps.BrewerMap* class. This function provides easy access to *BuPu* colormap. You can also use `get()` method to achieve the same result.

```
from SecretColors.cmaps import BrewerMap
import matplotlib
cm = BrewerMap(matplotlib)
plt.imshow(data, cm=cm.bu_pu()) # Or cm.get('BuPu')
plt.show()
```

Parameters

- **no_of_colors** – No of colors
- **is_qualitative** – If True, ColorMap will be qualitative
- **is_reversed** – If True, ColorMap will be reversed

Returns Matplotlib *LinearSegmentedColormap* or *ListedColormap* based on the options provided. You can use this directly in the matplotlib

bu_gn (*, *no_of_colors: int = None*, *is_qualitative: bool = False*, *is_reversed=False*)

This is special method available for *SecretColors.cmaps.BrewerMap* class. This function provides easy access to *BuGn* colormap. You can also use `get()` method to achieve the same result.

```
from SecretColors.cmaps import BrewerMap
import matplotlib
cm = BrewerMap(matplotlib)
plt.imshow(data, cm=cm.bu_gn()) # Or cm.get('BuGn')
plt.show()
```

Parameters

- **no_of_colors** – No of colors
- **is_qualitative** – If True, ColorMap will be qualitative
- **is_reversed** – If True, ColorMap will be reversed

Returns Matplotlib *LinearSegmentedColormap* or *ListedColormap* based on the options provided. You can use this directly in the matplotlib

yl_or_br (*, *no_of_colors: int = None*, *is_qualitative: bool = False*, *is_reversed=False*)

This is special method available for *SecretColors.cmaps.BrewerMap* class. This function provides easy access to *YlOrBr* colormap. You can also use `get()` method to achieve the same result.

```
from SecretColors.cmaps import BrewerMap
import matplotlib
cm = BrewerMap(matplotlib)
plt.imshow(data, cm=cm.yl_or_br()) # Or cm.get('YlOrBr')
plt.show()
```

Parameters

- **no_of_colors** – No of colors

- **is_qualitative** – If True, ColorMap will be qualitative
- **is_reversed** – If True, ColorMap will be reversed

Returns Matplotlib *LinearSegmentedColormap* or *ListedColormap* based on the options provided. You can use this directly in the matplotlib

yl_gn(*, *no_of_colors: int = None*, *is_qualitative: bool = False*, *is_reversed=False*)

This is special method available for *SecretColors.cmaps.BrewerMap* class. This function provides easy access to *YlGn* colormap. You can also use `get()` method to achieve the same result.

```
from SecretColors.cmaps import BrewerMap
import matplotlib
cm = BrewerMap(matplotlib)
plt.imshow(data, cm=cm.yl_gn()) # Or cm.get('YlGn')
plt.show()
```

Parameters

- **no_of_colors** – No of colors
- **is_qualitative** – If True, ColorMap will be qualitative
- **is_reversed** – If True, ColorMap will be reversed

Returns Matplotlib *LinearSegmentedColormap* or *ListedColormap* based on the options provided. You can use this directly in the matplotlib

rd_pu(*, *no_of_colors: int = None*, *is_qualitative: bool = False*, *is_reversed=False*)

This is special method available for *SecretColors.cmaps.BrewerMap* class. This function provides easy access to *RdPu* colormap. You can also use `get()` method to achieve the same result.

```
from SecretColors.cmaps import BrewerMap
import matplotlib
cm = BrewerMap(matplotlib)
plt.imshow(data, cm=cm.rd_pu()) # Or cm.get('RdPu')
plt.show()
```

Parameters

- **no_of_colors** – No of colors
- **is_qualitative** – If True, ColorMap will be qualitative
- **is_reversed** – If True, ColorMap will be reversed

Returns Matplotlib *LinearSegmentedColormap* or *ListedColormap* based on the options provided. You can use this directly in the matplotlib

yl_gn_bu(*, *no_of_colors: int = None*, *is_qualitative: bool = False*, *is_reversed=False*)

This is special method available for *SecretColors.cmaps.BrewerMap* class. This function provides easy access to *YlGnBu* colormap. You can also use `get()` method to achieve the same result.

```
from SecretColors.cmaps import BrewerMap
import matplotlib
cm = BrewerMap(matplotlib)
plt.imshow(data, cm=cm.yl_gn_bu()) # Or cm.get('YlGnBu')
plt.show()
```

Parameters

- **no_of_colors** – No of colors
- **is_qualitative** – If True, ColorMap will be qualitative
- **is_reversed** – If True, ColorMap will be reversed

Returns Matplotlib *LinearSegmentedColormap* or *ListedColormap* based on the options provided. You can use this directly in the matplotlib

gn_bu(*, *no_of_colors: int = None*, *is_qualitative: bool = False*, *is_reversed=False*)

This is special method available for *SecretColors.cmaps.BrewerMap* class. This function provides easy access to *GnBu* colormap. You can also use `get()` method to achieve the same result.

```
from SecretColors.cmaps import BrewerMap
import matplotlib
cm = BrewerMap(matplotlib)
plt.imshow(data, cm=cm.gn_bu()) # Or cm.get('GnBu')
plt.show()
```

Parameters

- **no_of_colors** – No of colors
- **is_qualitative** – If True, ColorMap will be qualitative
- **is_reversed** – If True, ColorMap will be reversed

Returns Matplotlib *LinearSegmentedColormap* or *ListedColormap* based on the options provided. You can use this directly in the matplotlib

yl_or_rd(*, *no_of_colors: int = 8*, *is_qualitative: bool = False*, *is_reversed=False*)

This is special method available for *SecretColors.cmaps.BrewerMap* class. This function provides easy access to *YlOrRd* colormap. You can also use `get()` method to achieve the same result.

```
from SecretColors.cmaps import BrewerMap
import matplotlib
cm = BrewerMap(matplotlib)
plt.imshow(data, cm=cm.yl_or_rd()) # Or cm.get('YlOrRd')
plt.show()
```

Parameters

- **no_of_colors** – No of colors
- **is_qualitative** – If True, ColorMap will be qualitative
- **is_reversed** – If True, ColorMap will be reversed

Returns Matplotlib *LinearSegmentedColormap* or *ListedColormap* based on the options provided. You can use this directly in the matplotlib

pu_rd(*, *no_of_colors: int = None*, *is_qualitative: bool = False*, *is_reversed=False*)

This is special method available for *SecretColors.cmaps.BrewerMap* class. This function provides easy access to *PuRd* colormap. You can also use `get()` method to achieve the same result.

```
from SecretColors.cmaps import BrewerMap
import matplotlib
cm = BrewerMap(matplotlib)
```

(continues on next page)

(continued from previous page)

```
plt.imshow(data, cm=cm.pu_rd()) # Or cm.get('PuRd')
plt.show()
```

Parameters

- **no_of_colors** – No of colors
- **is_qualitative** – If True, ColorMap will be qualitative
- **is_reversed** – If True, ColorMap will be reversed

Returns Matplotlib *LinearSegmentedColormap* or *ListedColormap* based on the options provided. You can use this directly in the matplotlib

pu_bu_gn(*, *no_of_colors: int = None*, *is_qualitative: bool = False*, *is_reversed=False*)

This is special method available for *SecretColors.cmaps.BrewerMap* class. This function provides easy access to *PuBuGn* colormap. You can also use `get()` method to achieve the same result.

```
from SecretColors.cmaps import BrewerMap
import matplotlib
cm = BrewerMap(matplotlib)
plt.imshow(data, cm=cm.pu_bu_gn()) # Or cm.get('PuBuGn')
plt.show()
```

Parameters

- **no_of_colors** – No of colors
- **is_qualitative** – If True, ColorMap will be qualitative
- **is_reversed** – If True, ColorMap will be reversed

Returns Matplotlib *LinearSegmentedColormap* or *ListedColormap* based on the options provided. You can use this directly in the matplotlib

3.4 TableauMap

class SecretColors.cmaps.**TableauMap**(*matplotlib*)

Tableau Color Maps are derived from Tableau Visualization Software. Unfortunately, their current version is not available publicly and hence we could not include here. Current color values are from their 9.x Legacy version which can be found here . This class acts as an wrapper around original colors which you can easily manipulate. This class is inherited from *ColorMapParent* and overrides few default methods which are available in our database.

Simplest way to use this class in your `matplotlib` workflow is following

```
import matplotlib
import matplotlib.pyplot as plt
from SecretColors.cmaps import TableauMap
tm = TableauMap(matplotlib)
plt.imshow(data, cmap=tm.tableau())
plt.show()
```

Here, *tableau()* is one of the available standard color map in this class. If you know the exact name of the color map, you can use following as well

```
plt.imshow(data, cmap= tm.get("Tableau")) # Alternate way
tm.get_all  # Gives you all available colormaps
tm.data # Returns available color data
```

To know which all color maps are available in current class you can use `get_all` attribute. These methods can be heavily customized. Take a look at *ColorMapParent* documentation.

List of available colormaps in TableauMap

- Tableau
- Tableau_medium
- Tableau_light
- Gray
- ColorBlind
- TrafficLight
- PurpleGray
- GreenOrange
- BlueRed
- Cyclic
- Green
- Blue
- Red
- Orange
- AquaRed
- AquaGreen
- AquaBrown
- RedGreen
- RedBlue
- RedBlack
- AreaRedGreen
- OrangeBlue
- GreenBlue
- RedWhiteGreen
- RedWhiteBlack
- OrangeWhiteBlue
- RedWhiteBlack_light
- OrangeWhiteBlue_light
- RedWhiteGreen_light
- RedGreen_light

See also:

SecretColors.cmaps.parent.ColorMapParent

data
Returns all available ColorMap data. This is valid ONLY for special subclass (e.g. BrewerMap). It will return None for 'ColorMap' class.

Return type dict

tableau(*, *no_of_colors: int = None*, *is_qualitative: bool = False*, *is_reversed=False*)
This is special method available for *SecretColors.cmaps.TableauMap* class. This function provides easy access to *Tableau* colormap. You can also use `get()` method to achieve the same result.

```
from SecretColors.cmaps import TableauMap
import matplotlib
cm = TableauMap(matplotlib)
plt.imshow(data, cm=cm.tableau()) # Or cm.get('Tableau')
plt.show()
```

Parameters

- **no_of_colors** – No of colors
- **is_qualitative** – If True, ColorMap will be qualitative
- **is_reversed** – If True, ColorMap will be reversed

Returns Matplotlib *LinearSegmentedColormap* or *ListedColormap* based on the options provided. You can use this directly in the matplotlib

tableau_light(*, *no_of_colors: int = None*, *is_qualitative: bool = False*, *is_reversed=False*)

This is special method available for *SecretColors.cmaps.TableauMap* class. This function provides easy access to *Tableau_light* colormap. You can also use `get()` method to achieve the same result.

```
from SecretColors.cmaps import TableauMap
import matplotlib
cm = TableauMap(matplotlib)
plt.imshow(data, cm=cm.tableau_light()) # Or cm.get('Tableau_light')
plt.show()
```

Parameters

- **no_of_colors** – No of colors
- **is_qualitative** – If True, ColorMap will be qualitative
- **is_reversed** – If True, ColorMap will be reversed

Returns Matplotlib *LinearSegmentedColormap* or *ListedColormap* based on the options provided. You can use this directly in the matplotlib

tableau_medium(*, *no_of_colors: int = None*, *is_qualitative: bool = False*, *is_reversed=False*)

This is special method available for *SecretColors.cmaps.TableauMap* class. This function provides easy access to *Tableau_medium* colormap. You can also use `get()` method to achieve the same result.

```
from SecretColors.cmaps import TableauMap
import matplotlib
cm = TableauMap(matplotlib)
plt.imshow(data, cm=cm.tableau_medium()) # Or cm.get('Tableau_medium')
plt.show()
```

Parameters

- **no_of_colors** – No of colors
- **is_qualitative** – If True, ColorMap will be qualitative
- **is_reversed** – If True, ColorMap will be reversed

Returns Matplotlib *LinearSegmentedColormap* or *ListedColormap* based on the options provided. You can use this directly in the matplotlib

colorblind(*, *no_of_colors: int = None*, *is_qualitative: bool = False*, *is_reversed=False*)

This is special method available for *SecretColors.cmaps.TableauMap* class. This function provides easy access to *ColorBlind* colormap. You can also use `get()` method to achieve the same result.

```
from SecretColors.cmaps import TableauMap
import matplotlib
cm = TableauMap(matplotlib)
plt.imshow(data, cm=cm.colorblind()) # Or cm.get('ColorBlind')
plt.show()
```

Parameters

- **no_of_colors** – No of colors
- **is_qualitative** – If True, ColorMap will be qualitative
- **is_reversed** – If True, ColorMap will be reversed

Returns Matplotlib *LinearSegmentedColormap* or *ListedColormap* based on the options provided. You can use this directly in the matplotlib

traffic_light(*, *no_of_colors: int = 9*, *is_qualitative: bool = False*, *is_reversed=False*)

This is special method available for *SecretColors.cmaps.TableauMap* class. This function provides easy access to *TrafficLight* colormap. You can also use `get()` method to achieve the same result.

```
from SecretColors.cmaps import TableauMap
import matplotlib
cm = TableauMap(matplotlib)
plt.imshow(data, cm=cm.traffic_light()) # Or cm.get('TrafficLight')
plt.show()
```

Parameters

- **no_of_colors** – No of colors
- **is_qualitative** – If True, ColorMap will be qualitative
- **is_reversed** – If True, ColorMap will be reversed

Returns Matplotlib *LinearSegmentedColormap* or *ListedColormap* based on the options provided. You can use this directly in the matplotlib

purple_gray(*, *no_of_colors: int = 12*, *is_qualitative: bool = False*, *is_reversed=False*)

This is special method available for *SecretColors.cmaps.TableauMap* class. This function provides easy access to *PurpleGray* colormap. You can also use `get()` method to achieve the same result.

```
from SecretColors.cmaps import TableauMap
import matplotlib
cm = TableauMap(matplotlib)
plt.imshow(data, cm=cm.purple_gray()) # Or cm.get('PurpleGray')
plt.show()
```

Parameters

- **no_of_colors** – No of colors
- **is_qualitative** – If True, ColorMap will be qualitative
- **is_reversed** – If True, ColorMap will be reversed

Returns Matplotlib *LinearSegmentedColormap* or *ListedColormap* based on the options provided. You can use this directly in the matplotlib

green_orange(*, *no_of_colors: int = 12*, *is_qualitative: bool = False*, *is_reversed=False*)

This is special method available for *SecretColors.cmaps.TableauMap* class. This function provides easy access to *GreenOrange* colormap. You can also use `get()` method to achieve the same result.

```
from SecretColors.cmaps import TableauMap
import matplotlib
cm = TableauMap(matplotlib)
plt.imshow(data, cm=cm.green_orange()) # Or cm.get('GreenOrange')
plt.show()
```

Parameters

- **no_of_colors** – No of colors
- **is_qualitative** – If True, ColorMap will be qualitative
- **is_reversed** – If True, ColorMap will be reversed

Returns Matplotlib *LinearSegmentedColormap* or *ListedColormap* based on the options provided. You can use this directly in the matplotlib

blue_red(*, *no_of_colors: int = 12*, *is_qualitative: bool = False*, *is_reversed=False*)

This is special method available for *SecretColors.cmaps.TableauMap* class. This function provides easy access to *BlueRed* colormap. You can also use `get()` method to achieve the same result.

```
from SecretColors.cmaps import TableauMap
import matplotlib
cm = TableauMap(matplotlib)
plt.imshow(data, cm=cm.blue_red()) # Or cm.get('BlueRed')
plt.show()
```

Parameters

- **no_of_colors** – No of colors
- **is_qualitative** – If True, ColorMap will be qualitative
- **is_reversed** – If True, ColorMap will be reversed

Returns Matplotlib *LinearSegmentedColormap* or *ListedColormap* based on the options provided. You can use this directly in the matplotlib

cyclic(*, *no_of_colors: int = 13*, *is_qualitative: bool = False*, *is_reversed=False*)

This is special method available for *SecretColors.cmaps.TableauMap* class. This function provides easy access to *Cyclic* colormap. You can also use `get()` method to achieve the same result.

```
from SecretColors.cmaps import TableauMap
import matplotlib
cm = TableauMap(matplotlib)
plt.imshow(data, cm=cm.cyclic()) # Or cm.get('Cyclic')
plt.show()
```

Parameters

- **no_of_colors** – No of colors

- **is_qualitative** – If True, ColorMap will be qualitative
- **is_reversed** – If True, ColorMap will be reversed

Returns Matplotlib *LinearSegmentedColormap* or *ListedColormap* based on the options provided. You can use this directly in the matplotlib

aqua_red(*, *no_of_colors: int = 11*, *is_qualitative: bool = False*, *is_reversed=False*)

This is special method available for *SecretColors.cmaps.TableauMap* class. This function provides easy access to *AquaRed* colormap. You can also use `get()` method to achieve the same result.

```
from SecretColors.cmaps import TableauMap
import matplotlib
cm = TableauMap(matplotlib)
plt.imshow(data, cm=cm.aqua_red()) # Or cm.get('AquaRed')
plt.show()
```

Parameters

- **no_of_colors** – No of colors
- **is_qualitative** – If True, ColorMap will be qualitative
- **is_reversed** – If True, ColorMap will be reversed

Returns Matplotlib *LinearSegmentedColormap* or *ListedColormap* based on the options provided. You can use this directly in the matplotlib

aqua_green(*, *no_of_colors: int = 11*, *is_qualitative: bool = False*, *is_reversed=False*)

This is special method available for *SecretColors.cmaps.TableauMap* class. This function provides easy access to *AquaGreen* colormap. You can also use `get()` method to achieve the same result.

```
from SecretColors.cmaps import TableauMap
import matplotlib
cm = TableauMap(matplotlib)
plt.imshow(data, cm=cm.aqua_green()) # Or cm.get('AquaGreen')
plt.show()
```

Parameters

- **no_of_colors** – No of colors
- **is_qualitative** – If True, ColorMap will be qualitative
- **is_reversed** – If True, ColorMap will be reversed

Returns Matplotlib *LinearSegmentedColormap* or *ListedColormap* based on the options provided. You can use this directly in the matplotlib

aqua_brown(*, *no_of_colors: int = 11*, *is_qualitative: bool = False*, *is_reversed=False*)

This is special method available for *SecretColors.cmaps.TableauMap* class. This function provides easy access to *AquaBrown* colormap. You can also use `get()` method to achieve the same result.

```
from SecretColors.cmaps import TableauMap
import matplotlib
cm = TableauMap(matplotlib)
plt.imshow(data, cm=cm.aqua_brown()) # Or cm.get('AquaBrown')
plt.show()
```

Parameters

- **no_of_colors** – No of colors
- **is_qualitative** – If True, ColorMap will be qualitative
- **is_reversed** – If True, ColorMap will be reversed

Returns Matplotlib *LinearSegmentedColormap* or *ListedColormap* based on the options provided. You can use this directly in the matplotlib

red_blue(*, *no_of_colors: int = 11*, *is_qualitative: bool = False*, *is_reversed=False*)

This is special method available for *SecretColors.cmaps.TableauMap* class. This function provides easy access to *RedBlue* colormap. You can also use `get()` method to achieve the same result.

```
from SecretColors.cmaps import TableauMap
import matplotlib
cm = TableauMap(matplotlib)
plt.imshow(data, cm=cm.red_blue()) # Or cm.get('RedBlue')
plt.show()
```

Parameters

- **no_of_colors** – No of colors
- **is_qualitative** – If True, ColorMap will be qualitative
- **is_reversed** – If True, ColorMap will be reversed

Returns Matplotlib *LinearSegmentedColormap* or *ListedColormap* based on the options provided. You can use this directly in the matplotlib

red_black(*, *no_of_colors: int = 11*, *is_qualitative: bool = False*, *is_reversed=False*)

This is special method available for *SecretColors.cmaps.TableauMap* class. This function provides easy access to *RedBlack* colormap. You can also use `get()` method to achieve the same result.

```
from SecretColors.cmaps import TableauMap
import matplotlib
cm = TableauMap(matplotlib)
plt.imshow(data, cm=cm.red_black()) # Or cm.get('RedBlack')
plt.show()
```

Parameters

- **no_of_colors** – No of colors
- **is_qualitative** – If True, ColorMap will be qualitative
- **is_reversed** – If True, ColorMap will be reversed

Returns Matplotlib *LinearSegmentedColormap* or *ListedColormap* based on the options provided. You can use this directly in the matplotlib

area_red_green(*, *no_of_colors: int = 21*, *is_qualitative: bool = False*, *is_reversed=False*)

This is special method available for *SecretColors.cmaps.TableauMap* class. This function provides easy access to *AreaRedGreen* colormap. You can also use `get()` method to achieve the same result.

```
from SecretColors.cmaps import TableauMap
import matplotlib
cm = TableauMap(matplotlib)
```

(continues on next page)

(continued from previous page)

```
plt.imshow(data, cm=cm.area_red_green()) # Or cm.get('AreaRedGreen')
plt.show()
```

Parameters

- **no_of_colors** – No of colors
- **is_qualitative** – If True, ColorMap will be qualitative
- **is_reversed** – If True, ColorMap will be reversed

Returns Matplotlib *LinearSegmentedColormap* or *ListedColormap* based on the options provided. You can use this directly in the matplotlib

orange_blue(*, *no_of_colors: int = 13*, *is_qualitative: bool = False*, *is_reversed=False*)

This is special method available for *SecretColors.cmaps.TableauMap* class. This function provides easy access to *OrangeBlue* colormap. You can also use `get()` method to achieve the same result.

```
from SecretColors.cmaps import TableauMap
import matplotlib
cm = TableauMap(matplotlib)
plt.imshow(data, cm=cm.orange_blue()) # Or cm.get('OrangeBlue')
plt.show()
```

Parameters

- **no_of_colors** – No of colors
- **is_qualitative** – If True, ColorMap will be qualitative
- **is_reversed** – If True, ColorMap will be reversed

Returns Matplotlib *LinearSegmentedColormap* or *ListedColormap* based on the options provided. You can use this directly in the matplotlib

green_blue(*, *no_of_colors: int = 11*, *is_qualitative: bool = False*, *is_reversed=False*)

This is special method available for *SecretColors.cmaps.TableauMap* class. This function provides easy access to *GreenBlue* colormap. You can also use `get()` method to achieve the same result.

```
from SecretColors.cmaps import TableauMap
import matplotlib
cm = TableauMap(matplotlib)
plt.imshow(data, cm=cm.green_blue()) # Or cm.get('GreenBlue')
plt.show()
```

Parameters

- **no_of_colors** – No of colors
- **is_qualitative** – If True, ColorMap will be qualitative
- **is_reversed** – If True, ColorMap will be reversed

Returns Matplotlib *LinearSegmentedColormap* or *ListedColormap* based on the options provided. You can use this directly in the matplotlib

red_white_green(*, *no_of_colors: int = 11*, *is_qualitative: bool = False*, *is_reversed=False*)
This is special method available for *SecretColors.cmaps.TableauMap* class. This function provides easy access to *RedWhiteGreen* colormap. You can also use `get()` method to achieve the same result.

```
from SecretColors.cmaps import TableauMap
import matplotlib
cm = TableauMap(matplotlib)
plt.imshow(data, cm=cm.red_white_green()) # Or cm.get('RedWhiteGreen')
plt.show()
```

Parameters

- **no_of_colors** – No of colors
- **is_qualitative** – If True, ColorMap will be qualitative
- **is_reversed** – If True, ColorMap will be reversed

Returns Matplotlib *LinearSegmentedColormap* or *ListedColormap* based on the options provided. You can use this directly in the matplotlib

red_white_black(*, *no_of_colors: int = 11*, *is_qualitative: bool = False*, *is_reversed=False*)
This is special method available for *SecretColors.cmaps.TableauMap* class. This function provides easy access to *RedWhiteBlack* colormap. You can also use `get()` method to achieve the same result.

```
from SecretColors.cmaps import TableauMap
import matplotlib
cm = TableauMap(matplotlib)
plt.imshow(data, cm=cm.red_white_black()) # Or cm.get('RedWhiteBlack')
plt.show()
```

Parameters

- **no_of_colors** – No of colors
- **is_qualitative** – If True, ColorMap will be qualitative
- **is_reversed** – If True, ColorMap will be reversed

Returns Matplotlib *LinearSegmentedColormap* or *ListedColormap* based on the options provided. You can use this directly in the matplotlib

orange_white_blue(*, *no_of_colors: int = 11*, *is_qualitative: bool = False*, *is_reversed=False*)
This is special method available for *SecretColors.cmaps.TableauMap* class. This function provides easy access to *OrangeWhiteBlue* colormap. You can also use `get()` method to achieve the same result.

```
from SecretColors.cmaps import TableauMap
import matplotlib
cm = TableauMap(matplotlib)
plt.imshow(data, cm=cm.orange_white_blue()) # Or cm.get('OrangeWhiteBlue')
plt.show()
```

Parameters

- **no_of_colors** – No of colors

- **is_qualitative** – If True, ColorMap will be qualitative
- **is_reversed** – If True, ColorMap will be reversed

Returns Matplotlib *LinearSegmentedColormap* or *ListedColormap* based on the options provided. You can use this directly in the matplotlib

red_white_black_light(*, *no_of_colors: int = 10*, *is_qualitative: bool = False*, *is_reversed=False*)

This is special method available for *SecretColors.cmaps.TableauMap* class. This function provides easy access to *RedWhiteBlack_light* colormap. You can also use `get()` method to achieve the same result.

```
from SecretColors.cmaps import TableauMap
import matplotlib
cm = TableauMap(matplotlib)
plt.imshow(data, cm=cm.red_white_black_light()) # Or cm.get('RedWhiteBlack_
↪light')
plt.show()
```

Parameters

- **no_of_colors** – No of colors
- **is_qualitative** – If True, ColorMap will be qualitative
- **is_reversed** – If True, ColorMap will be reversed

Returns Matplotlib *LinearSegmentedColormap* or *ListedColormap* based on the options provided. You can use this directly in the matplotlib

orange_white_blue_light(*, *no_of_colors: int = 11*, *is_qualitative: bool = False*, *is_reversed=False*)

This is special method available for *SecretColors.cmaps.TableauMap* class. This function provides easy access to *OrangeWhiteBlue_light* colormap. You can also use `get()` method to achieve the same result.

```
from SecretColors.cmaps import TableauMap
import matplotlib
cm = TableauMap(matplotlib)
plt.imshow(data, cm=cm.orange_white_blue_light()) # Or cm.get(
↪'OrangeWhiteBlue_light')
plt.show()
```

Parameters

- **no_of_colors** – No of colors
- **is_qualitative** – If True, ColorMap will be qualitative
- **is_reversed** – If True, ColorMap will be reversed

Returns Matplotlib *LinearSegmentedColormap* or *ListedColormap* based on the options provided. You can use this directly in the matplotlib

red_white_green_light(*, *no_of_colors: int = 11*, *is_qualitative: bool = False*, *is_reversed=False*)

This is special method available for *SecretColors.cmaps.TableauMap* class. This function provides easy access to *RedWhiteGreen_light* colormap. You can also use `get()` method to achieve the same result.

```
from SecretColors.cmaps import TableauMap
import matplotlib
cm = TableauMap(matplotlib)
plt.imshow(data, cm=cm.red_white_green_light()) # Or cm.get('RedWhiteGreen_
↪light')
plt.show()
```

Parameters

- **no_of_colors** – No of colors
- **is_qualitative** – If True, ColorMap will be qualitative
- **is_reversed** – If True, ColorMap will be reversed

Returns Matplotlib *LinearSegmentedColormap* or *ListedColormap* based on the options provided. You can use this directly in the matplotlib

red_green_light(*, *no_of_colors: int = 11*, *is_qualitative: bool = False*, *is_reversed=False*)

This is special method available for *SecretColors.cmaps.TableauMap* class. This function provides easy access to *RedGreen_light* colormap. You can also use `get()` method to achieve the same result.

```
from SecretColors.cmaps import TableauMap
import matplotlib
cm = TableauMap(matplotlib)
plt.imshow(data, cm=cm.red_green_light()) # Or cm.get('RedGreen_light')
plt.show()
```

Parameters

- **no_of_colors** – No of colors
- **is_qualitative** – If True, ColorMap will be qualitative
- **is_reversed** – If True, ColorMap will be reversed

Returns Matplotlib *LinearSegmentedColormap* or *ListedColormap* based on the options provided. You can use this directly in the matplotlib

CHAPTER 4

Utils

SecretColors.utils.**rgb_to_cmy**(*r*, *g*, *b*) → tuple
Converts RGB to CMY (both between 0-1)

Parameters

- **r** – Red
- **g** – Green
- **b** – Blue

Returns Cyan, Magenta, Yellow

SecretColors.utils.**cmy_to_rgb**(*c*, *m*, *y*) → tuple
Converts CMY to RGB (both between 0-1)

Parameters

- **c** – Cyan
- **m** – Magenta
- **y** – Yellow

Returns Red, Green, Blue

SecretColors.utils.**cmy_to_cmyk**(*c*, *m*, *y*) → tuple
Converts CMY to CMYK (both between 0-1)

Parameters

- **c** – Cyan
- **m** – Magenta
- **y** – Yellow

Returns Cyan, Magenta, Yellow, Black

SecretColors.utils.**cmyk_to_cmy**(*c*, *m*, *y*, *k*)
Converts CMYK to CMY (both between 0-1)

Parameters

- **c** – Cyan
- **m** – Magenta
- **y** – Yellow
- **k** – Black

Returns Cyan, Magenta, Black

`SecretColors.utils.`**`rgb_to_hsv`**(*r*, *g*, *b*)
Converts RGB to HSV

Hue will be normalized and will be on the scale of 0-1 than 0-360 Conversion formula is taken from https://www.rapidtables.com/convert/color/rgb-to-hsv.html

Parameters

- **r** – Red (0 to 1)
- **g** – Green (0 to 1)
- **b** – Blue (0 to 1)

Returns (Hue, Saturation, Lightness) on the scale of (0 to 1)

`SecretColors.utils.`**`hsv_to_rgb`**(*h*, *s*, *v*) → tuple
Converts HSV to RGB (both between 0-1)

Conversion calculation from: https://www.rapidtables.com/convert/color/hsv-to-rgb.html

Parameters

- **h** – Hue
- **s** – Saturation
- **v** – Value

Returns Red, Green, Blue

`SecretColors.utils.`**`rgb_to_hsl`**(*r*, *g*, *b*) → tuple
Converts RGB tuple into HSL tuple Calculations are taken from http://www.easyrgb.com/en/math.php

Parameters

- **r** – Red (between 0 to 1)
- **g** – Green (between 0 to 1)
- **b** – Blue (between 0 to 1)

Returns (hue, saturation, lightness) All between 0 to 1

`SecretColors.utils.`**`hsl_to_rgb`**(*h*, *s*, *l*) → tuple
Converts HSL values to RGB tuple. Calculations are taken from http://www.easyrgb.com/en/math.php

Parameters

- **h** – Hue (between 0 to 1)
- **s** – Saturation (between 0 to 1)
- **l** – Lightness (between 0 to 1)

Returns (Red, Green, Blue) between 0 to 1

SecretColors.utils.**rgb_to_rgb255**(*r: float*, *g: float*, *b: float*) → Tuple[float, float, float]
Converts 0-1 based RGB into 0-255 based RGB

Parameters

- **r** – Red (between 0-1)
- **g** – Green (between 0-1)
- **b** – Blue (between 0-1)

Returns Red, Green, Blue (between 0-255)

SecretColors.utils.**rgb255_to_hex**(*r: int*, *g: int*, *b: int*) → str
Converts 0-255 based RGB to Hex

Parameters

- **r** – Red (between 0-255)
- **g** – Green (between 0-255)
- **b** – Blue (between 0-255)

Returns Hex color code

SecretColors.utils.**rgb255_to_rgb**(*r: int*, *g: int*, *b: int*) → Tuple[float, float, float]
Converts 0-255 based RGB to 0-1 based RGB

Parameters

- **r** – Red (between 0-255)
- **g** – Green (between 0-255)
- **b** – Blue (between 0-255)

Returns Red, Green, Blue (between 0-1)

SecretColors.utils.**rgb255_to_hsv**(*r: int*, *g: int*, *b: int*)
Converts 0-255 based RGB to HSV (0-1 based)

Parameters

- **r** – Red (between 0-255)
- **g** – Green (between 0-255)
- **b** – Blue (between 0-255)

Returns Hue, Saturation, Value (0-1 based)

SecretColors.utils.**rgb255_to_hsl**(*r: int*, *g: int*, *b: int*)
Converts 0-255 based RGB to HSL (0-1 based)

Parameters

- **r** – Red (between 0-255)
- **g** – Green (between 0-255)
- **b** – Blue (between 0-255)

Returns Hue, Saturation, Lightness (0-1 based)

SecretColors.utils.**hex_to_rgb**(*hex_string: str*)
Converts Hex to RGB (0-1)

Parameters **hex_string** – Hex string

Returns Red, Green, Blue (between 0-1)

SecretColors.utils.**rgb_to_hsb**(*r*, *g*, *b*)
Converts RGB to HSB (both between 1-0)

Parameters

- **r** – Red
- **g** – Green
- **b** – Blue

Returns Hue, Saturation, Brightness (between 0-1)

SecretColors.utils.**hsb_to_rgb**(*h*, *s*, *b*)
Converts HSB to RGB (both between 0-1)

Parameters

- **h** – Hue
- **s** – Saturation
- **b** – Brightness

Returns Red, Green, Blue (between 0-1)

SecretColors.utils.**apply_gamma_transform**(*value*)
Transforms values from linear scale to non-linea by applying gamma transform.

Parameters **value** – Values to be transformed

Returns Transformed values

SecretColors.utils.**apply_linear_transform**(*value*)
Transforms value from non-linear scale (with gamma transform) to linear

Parameters **value** – Value to be transform (between 0-1)

Returns Transformed value (between 0-1)

SecretColors.utils.**rgb_to_srgb**(*r*, *g*, *b*)
Converts RGB to sRGB

Note: This function is still in beta-testing. Do not use in your production code.

Parameters

- **r** – Red
- **g** – Green
- **b** – Blue

Returns sRed, sGreen, sBlue

SecretColors.utils.**srgb_to_rgb**(*sr*, *sg*, *sb*)
Converts sRGB to RGB

Note: This function is still in beta-testing. Do not use in your production code.

Parameters

- **sr** – sRed
- **sg** – sGreen
- **sb** – sBlue

Returns Red, Green, Blue

SecretColors.utils.**rgb_to_xyz**(*r*, *g*, *b*, ***, *reference='D65'*, *clip=True*)
Converts Linear-RGB (0-1) to CIE-XYZ

Note: This function is still in beta-testing. Do not use in your production code.

Parameters

- **r** – Red
- **g** – Green
- **b** – Blue
- **reference** – White reference (default: D65)
- **clip** – If True, values below 0 and above 1 will be clipped

Returns CIE-X, CIE-Y, CIE-Z

SecretColors.utils.**xyz_to_rgb**(*x*, *y*, *z*, ***, *reference='D65'*, *clip=True*)
Converts CIE-XYZ to Linear-RGB (0-1)

Note: This function is still in beta-testing. Do not use in your production code.

Parameters

- **x** – CIE-X
- **y** – CIE-Y
- **z** – CIE-Z
- **reference** – White reference (default: D65)
- **clip** – If True, values below 0 and above 1 will be clipped

Returns Red, Green Blue

SecretColors.utils.**adobe_rgb_to_xyz**(*r*, *g*, *b*, ***, *reference='D65'*, *clip=True*)

Converts adobe-RGB to CIE-XYZ

Note: This function is still in beta-testing. Do not use in your production code.

param r adobe-Red

param g adobe-Green

param b adobe-Blue

param reference White reference (default: D65)

param clip If True, values above 1 and below 0 will be clipped

return CIE-X, CIE-Y, CIE-Z

SecretColors.utils.**xyz_to_adobe_rgb**(*x*, *y*, *z*, ***, *reference='D65'*, *clip=True*)
Converts CIE-XYZ to adobe-RGB

Note: This function is still in beta-testing. Do not use in your production code.

Parameters

- **x** – CIE-X
- **y** – CIE-Y
- **z** – CIE-Z

- **reference** – White reference (default: D65)
- **clip** – If True, values below 0 and above 1 will be clipped

Returns adobe-Red, adobe-Green adobe-Blue

SecretColors.utils.**relative_luminance**(*hex_color: str*)
Relative luminance according to WCAG 2.0 standard https://www.w3.org/WAI/GL/wiki/Relative_luminance :param hex_color: :return:

SecretColors.utils.**text_color**(*hex_color: str*)
Provides black or white color which can be used for text on given hex color background

According to WCA 2.0 standards https://www.w3.org/WAI/GL/wiki/Relative_luminance and https://medium.muz.li/the-science-of-color-contrast-an-expert-designers-guide-33e84c41d156

```
>>> text_color("#ffffff") # '#000000'
```

Parameters **hex_color** – background color

Returns proper text color

SecretColors.utils.**color_in_between**(*c1*, *c2*, *no_of_colors=1*) → list
Creates color between two colors in RGB ColorSpace

```
>>> color_in_between("#fb4b53", "#408bfc") # ['#9d6aa7']
>>> color_in_between("#fb4b53", "#408bfc", 3) # ['#cc5a7d', '#9d6aa7', '#6e7ad1']
```

Parameters

- **c1** – Hex of first color
- **c2** – Hex of second color
- **no_of_colors** – How many colors in between? [Default :1]

Returns List of Colors between provided colors in RGB space

SecretColors.utils.**hex_to_hsl**(*hex_string*)
Converts HEX to HSL (0-1)

Parameters **hex_string** – Hex String

Returns Heu, Saturation, Lightness (between 0-1)

SecretColors.utils.**hsl_to_hex**(*h*, *s*, *l*)
Converts HSL (between 0-1) into Hex string

Parameters

- **h** – Hue
- **s** – Saturation
- **l** – Lightness

Returns Hex String

SecretColors.utils.**get_complementary**(*hex_color: str*)
Returns complementary color

```
>>> get_complementary("#fb4b53") # '#4afaf3'
```

Parameters **hex_color** – Hex color

Returns Complementary Hex color

SecretColors.utils.**simulate_green_blindness**(*r*, *g*, *b*)

Adjust RGB values such that it can 'simulate' Green blindness

Conversion formula taken from https://personal.sron.nl/~pault/#sec:colour_blindness

Note: This function is still in beta-testing. Do not use in your production code.

Parameters

- **r** – Red
- **g** – Green
- **b** – Blue

Returns Red, Green, Blue (seen by Green-Blind person)

SecretColors.utils.**simulate_red_blindness**(*r*, *g*, *b*)

Adjust RGB values such that it can 'simulate' Red blindness

Conversion formula taken from https://personal.sron.nl/~pault/#sec:colour_blindness

Note: This function is still in beta-testing. Do not use in your production code.

Parameters

- **r** – Red
- **g** – Green
- **b** – Blue

Returns Red, Green, Blue (seen by Red-Blind person)

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

SecretColors.utils, 53

Index

Symbols

__init__() (*SecretColors.ColorWheel method*), 24
__init__() (*SecretColors.Palette method*), 4
__init__() (*SecretColors.cmaps.parent.ColorMapParent method*), 27

A

accent() (*SecretColors.cmaps.BrewerMap method*), 36
adobe_rgb_to_xyz() (*in module SecretColors.utils*), 57
amber() (*SecretColors.Palette method*), 20
analogous() (*SecretColors.ColorWheel method*), 24
apply_gamma_transform() (*in module SecretColors.utils*), 56
apply_linear_transform() (*in module SecretColors.utils*), 56
aqua() (*SecretColors.Palette method*), 15
aqua_brown() (*SecretColors.cmaps.TableauMap method*), 47
aqua_green() (*SecretColors.cmaps.TableauMap method*), 47
aqua_red() (*SecretColors.cmaps.TableauMap method*), 47
area_red_green() (*SecretColors.cmaps.TableauMap method*), 48

B

black() (*SecretColors.Palette method*), 13
blue() (*SecretColors.Palette method*), 8
blue_light() (*SecretColors.Palette method*), 19
blue_red() (*SecretColors.cmaps.TableauMap method*), 46
br_bg() (*SecretColors.cmaps.BrewerMap method*), 34
BrewerMap (*class in SecretColors.cmaps*), 31
brown() (*SecretColors.Palette method*), 21
bu_gn() (*SecretColors.cmaps.BrewerMap method*), 39
bu_pu() (*SecretColors.cmaps.BrewerMap method*), 39

C

cerulean() (*SecretColors.Palette method*), 15
cmap_from() (*SecretColors.Palette static method*), 6
cmy_to_cmyk() (*in module SecretColors.utils*), 53
cmy_to_rgb() (*in module SecretColors.utils*), 53
cmyk_to_cmy() (*in module SecretColors.utils*), 53
color (*SecretColors.ColorWheel attribute*), 24
color_in_between() (*in module SecretColors.utils*), 58
colorblind() (*SecretColors.cmaps.TableauMap method*), 44
ColorMap (*class in SecretColors.cmaps*), 29
ColorMapParent (*class in SecretColors.cmaps.parent*), 27
colors (*SecretColors.Palette attribute*), 5
ColorWheel (*class in SecretColors*), 23
complementary() (*SecretColors.ColorWheel method*), 24
creator_url (*SecretColors.Palette attribute*), 4
cyan() (*SecretColors.Palette method*), 10
cycle() (*SecretColors.Palette method*), 4
cyclic() (*SecretColors.cmaps.TableauMap method*), 46

D

dark2() (*SecretColors.cmaps.BrewerMap method*), 36
data (*SecretColors.cmaps.BrewerMap attribute*), 32
data (*SecretColors.cmaps.ColorMap attribute*), 30
data (*SecretColors.cmaps.parent.ColorMapParent attribute*), 28
data (*SecretColors.cmaps.TableauMap attribute*), 43

F

from_list() (*SecretColors.cmaps.parent.ColorMapParent method*), 28

G

get() (*SecretColors.cmaps.parent.ColorMapParent method*), 29

get() (*SecretColors.Palette method*), 7
get_all (*SecretColors.cmaps.parent.ColorMapParent attribute*), 28
get_color_dict (*SecretColors.Palette attribute*), 4
get_color_list (*SecretColors.Palette attribute*), 4
get_colors() (*SecretColors.cmaps.parent.ColorMapParent method*), 28
get_complementary() (*in module SecretColors.utils*), 58
gn_bu() (*SecretColors.cmaps.BrewerMap method*), 41
gold() (*SecretColors.Palette method*), 16
gray() (*SecretColors.Palette method*), 12
gray_blue() (*SecretColors.Palette method*), 22
gray_cool() (*SecretColors.Palette method*), 11
gray_neutral() (*SecretColors.Palette method*), 12
gray_warm() (*SecretColors.Palette method*), 12
green() (*SecretColors.Palette method*), 9
green_blue() (*SecretColors.cmaps.TableauMap method*), 49
green_light() (*SecretColors.Palette method*), 20
green_orange() (*SecretColors.cmaps.TableauMap method*), 46

H

hex_to_hsl() (*in module SecretColors.utils*), 58
hex_to_rgb() (*in module SecretColors.utils*), 55
hsb_to_rgb() (*in module SecretColors.utils*), 56
hsl_to_hex() (*in module SecretColors.utils*), 58
hsl_to_rgb() (*in module SecretColors.utils*), 54
hsv_to_rgb() (*in module SecretColors.utils*), 54

I

indigo() (*SecretColors.Palette method*), 18

L

lime() (*SecretColors.Palette method*), 15

M

magenta() (*SecretColors.Palette method*), 9
make_darker() (*SecretColors.ColorWheel method*), 24
make_lighter() (*SecretColors.ColorWheel method*), 24

N

name (*SecretColors.Palette attribute*), 4

O

or_rd() (*SecretColors.cmaps.BrewerMap method*), 38
orange() (*SecretColors.Palette method*), 17
orange_blue() (*SecretColors.cmaps.TableauMap method*), 49
orange_deep() (*SecretColors.Palette method*), 21
orange_white_blue() (*SecretColors.cmaps.TableauMap method*), 50
orange_white_blue_light() (*SecretColors.cmaps.TableauMap method*), 51

P

paired() (*SecretColors.cmaps.BrewerMap method*), 37
Palette (*class in SecretColors*), 3
palette (*SecretColors.cmaps.parent.ColorMapParent attribute*), 28
pastel1() (*SecretColors.cmaps.BrewerMap method*), 37
pastel2() (*SecretColors.cmaps.BrewerMap method*), 37
peach() (*SecretColors.Palette method*), 17
pi_yg() (*SecretColors.cmaps.BrewerMap method*), 33
pink() (*SecretColors.Palette method*), 19
pr_gn() (*SecretColors.cmaps.BrewerMap method*), 33
pu_bu() (*SecretColors.cmaps.BrewerMap method*), 38
pu_bu_gn() (*SecretColors.cmaps.BrewerMap method*), 42
pu_or() (*SecretColors.cmaps.BrewerMap method*), 35
pu_rd() (*SecretColors.cmaps.BrewerMap method*), 41
purple() (*SecretColors.Palette method*), 10
purple_deep() (*SecretColors.Palette method*), 19
purple_gray() (*SecretColors.cmaps.TableauMap method*), 45

R

random() (*SecretColors.Palette method*), 5
random_balanced() (*SecretColors.Palette method*), 6
random_gradient() (*SecretColors.Palette method*), 6
rd_bu() (*SecretColors.cmaps.BrewerMap method*), 33
rd_gy() (*SecretColors.cmaps.BrewerMap method*), 34
rd_pu() (*SecretColors.cmaps.BrewerMap method*), 40
rd_yl_bu() (*SecretColors.cmaps.BrewerMap method*), 34
rd_yl_gn() (*SecretColors.cmaps.BrewerMap method*), 32
red() (*SecretColors.Palette method*), 8
red_black() (*SecretColors.cmaps.TableauMap method*), 48
red_blue() (*SecretColors.cmaps.TableauMap method*), 48
red_green_light() (*SecretColors.cmaps.TableauMap method*), 52
red_orange() (*SecretColors.Palette method*), 13
red_white_black() (*SecretColors.cmaps.TableauMap method*), 50

red_white_black_light() (SecretColors.cmaps.TableauMap method), 51
red_white_green() (SecretColors.cmaps.TableauMap method), 49
red_white_green_light() (SecretColors.cmaps.TableauMap method), 51
relative_luminance() (in module SecretColors.utils), 58
reset() (SecretColors.ColorWheel method), 24
rgb255_to_hex() (in module SecretColors.utils), 55
rgb255_to_hsl() (in module SecretColors.utils), 55
rgb255_to_hsv() (in module SecretColors.utils), 55
rgb255_to_rgb() (in module SecretColors.utils), 55
rgb_to_cmy() (in module SecretColors.utils), 53
rgb_to_hsb() (in module SecretColors.utils), 56
rgb_to_hsl() (in module SecretColors.utils), 54
rgb_to_hsv() (in module SecretColors.utils), 54
rgb_to_rgb255() (in module SecretColors.utils), 54
rgb_to_srgb() (in module SecretColors.utils), 56
rgb_to_xyz() (in module SecretColors.utils), 57
rotate_hue() (SecretColors.ColorWheel method), 24
rotate_lightness() (SecretColors.ColorWheel method), 24
rotate_saturation() (SecretColors.ColorWheel method), 24

S

SecretColors.utils (module), 53
seed (SecretColors.Palette attribute), 5
set1() (SecretColors.cmaps.BrewerMap method), 35
set2() (SecretColors.cmaps.BrewerMap method), 35
set3() (SecretColors.cmaps.BrewerMap method), 36
simulate_green_blindness() (in module SecretColors.utils), 59
simulate_red_blindness() (in module SecretColors.utils), 59
spectral() (SecretColors.cmaps.BrewerMap method), 32
srgb_to_rgb() (in module SecretColors.utils), 56

T

tableau() (SecretColors.cmaps.TableauMap method), 43
tableau_light() (SecretColors.cmaps.TableauMap method), 44
tableau_medium() (SecretColors.cmaps.TableauMap method), 44
TableauMap (class in SecretColors.cmaps), 42
teal() (SecretColors.Palette method), 11
tetradic() (SecretColors.ColorWheel method), 24
text_color() (in module SecretColors.utils), 58
text_color() (SecretColors.ColorWheel method), 25
traffic_light() (SecretColors.cmaps.TableauMap method), 45
triadic() (SecretColors.ColorWheel method), 24

U

ultramarine() (SecretColors.Palette method), 14

V

version (SecretColors.Palette attribute), 4
violet() (SecretColors.Palette method), 18

W

white() (SecretColors.Palette method), 14

X

xyz_to_adobe_rgb() (in module SecretColors.utils), 57
xyz_to_rgb() (in module SecretColors.utils), 57

Y

yellow() (SecretColors.Palette method), 16
yl_gn() (SecretColors.cmaps.BrewerMap method), 40
yl_gn_bu() (SecretColors.cmaps.BrewerMap method), 40
yl_or_br() (SecretColors.cmaps.BrewerMap method), 39
yl_or_rd() (SecretColors.cmaps.BrewerMap method), 41