colorwheels

Release 0.7.3

Quantum Spaces

Sep 06, 2021

GETTING STARTED

CHAPTER

ONE

COLORWHEELS - AN ENDLESS COLOR GENERATOR

This project is a continuous color generator for Python. The idea is to continuously serve a sequence of colors to your application, in any desired format.

We use the package for easy color handling in our electronics projects on Raspberry Pi (RGB LEDs, RGB Panels and other). This doesn't limit the use of course, use it anywhere you like!

The idea behind is an endless colorwheel for photographers - the wheel continuously turns around to generate the next color - and ideally, colors have a continuity when starting all over.

We have also bundled an easy configuration handling, via YAML files (see *YAML Color Definitions*). Define your color palettes and patterns in a script file and separate color definitions from code. Then re-use. Apart from standard color sequences, we've added color generators - as of this writing, Rainbow effect generators are available.

If you're wondering how hard is to use colorwheels on a keybow (shown above), here's the code snippet - this loop ensures a rainbow effect on a button, which awaits a keypress.

```
# wheel is initialized and definitions loaded or generated
while True:
   color = wheel.next()
   keybow.set_led(9, color[0], color[1], color[2])
   keybow.show()
   time.sleep(0.1)
```

We have a small tutorial available as well. Go to the section *Using Colorwheels* to get started.

1.1 Using Colorwheels

1.1.1 Introduction

This is a getting started guide, on using Colorwheels modules. As mentioned in the introduction, a *Colorwheels* instance is an endless color sequence generator to drive changing colors in our RGB Led hardware.

Once you have covered the basics, check our tutorial page on *Color Handling*.

1.1.2 Importing

To import all `colorwheels` modules, simply add an import statement to your script.

```
import colorwheels
```

1.1.3 Using the generator

We'll start using our generator with preset colors. Learn more in the next sections about changing colors and other features. For now, we'll simply use default settings.

The pre-defined generator endlessly rotates red, green and blue. By default, it also returns these colors in an RGB tuple. You can however get colors in other formats as well.

A minimal generator code could look like this:

```
# mywheels.py
#
# Colorwheel Generator Example 1

import colorwheels

wheels = colorwheels.Colorwheels()

for i in range(5):
    print(next(wheels))
```

Your output can be similar to the below (note, if you log to console, there could be more output lines from the logger):

```
$ python mywheels.py
(255, 0, 0)
(0, 255, 0)
(0, 0, 255)
(255, 0, 0)
(0, 255, 0)
$
```

Similarly, you can invoke one of several 'next' methods directly. See below:

```
# ...
for i in range(5):
    print(wheels.next())
```

1.1.4 Changing Color Type

The object returns different data formats, based on a type setting. You can receive RGB, RGBA or hexadecimal values per each iteration.

The following example iterates colors with hexadecimal output.

```
# mywheels.py
#
# Colorwheel Generator Example 2
```

(continues on next page)

(continued from previous page)

```
import colorwheels

wheels = colorwheels.Colorwheels()

# set generator type ("rgb_tuple" or "rgba_tuple" or "hexadecimal")
wheels.set_generator_type("hexadecimal")

for i in range(5):
    print(next(wheels))
```

With the below output:

```
$ python mywheels.py
#ff0000
#00ff00
#0000ff
#ff0000
#00ff00
$
```

Check the next section to see how to work with colors: *Color Handling*

1.1.5 Switching colorwheels

You can switch palettes on the run. Below is a more complete example of real-life usage. Say you have a button with an RGB led, and you want to rotate a few red tints when button is pressed, otherwise animate a green palette, if released. The trick is in the `activate_colorwheel` method, which locates a wheel by name and activates it.

```
# mywheels.py
#
# Colorwheel Generator Example 3

import colorwheels
import time

wheels = colorwheels.Colorwheels()

# load your color palettes here. For example 'reds' for red tints,
# 'greens' for green tints

def button_pressed(self):
    # do some logic here, return True or False
    return True

while(True):
    if button_pressed:
        wheels.activate_colorwheel("reds")
    else:
        wheels.active_wheel("greens")
```

(continues on next page)

(continued from previous page)

```
    color = next(wheels)
    # apply color to button / LED etc.
    time.sleep(1)
```

1.1.6 Where to next?

There are other features where our generator will help you. Check the available functions of *Colorwheels* and *ColorwheelsConfig*, as well as the features in our color classes: *ColorItem* and *WheelItem*.

Below is an example of using a so far not-mentioned class method of *WheelItem* - `wheel_complement`: as you'll find out in the documentation, the method copies over an existing wheel item and switches all colors to complementing colors. Comes in very useful for example, when you want to handle object colors with a well contrasting background.

```
# mywheels.py
#
# Colorwheel Generator Example 4

import colorwheels
import time

text_wheel = colorwheels.Colorwheels()
background_wheel = colorwheels.Colorwheels()

# load your color palettes here. For example 'reds' for red tints,
# 'greens' for green tints. We're going to activate blue tints.

text_wheel.activate_colorwheel("blues")
# background wheel generates a new palette 'blues_complement'
# the palette is local (i.e. not stored in the config singleton)
background_wheel.active_wheel = colorwheels.WheelItem.wheel_complement(text_wheel.active_
↪wheel)

while(True):
    text_color = next(text_wheel)
    background_color = next(background_wheel)
    # write your text with obtained colors
    time.sleep(1)
```

1.1.7 Logging

The package uses standard Python logging. Please configure according to your needs.

1.2 Color Handling

1.2.1 Introduction

The default color configuration comes with one RGB sequence as a default in the configuration file. You can import color sequences from a YAML definition file as well, or you can define your own in code of course. This section describes how.

Get more details on crafting your YAML file in this tutorial section: *YAML Color Definitions*.

1.2.2 Color Objects

Colorwheels introduces 2 objects to handle color definitions: *ColorItem*, *WheelItem* for one specific color generator. The base class - *Colorwheels* - handles multiple generator defintions, and related operations: loading, swapping color schemes etc.

- *ColorItem* : this is a dataclass which captures RGB information, and handles color conversions. You can use it stand-alone, or you can use it in lists of colors. One of those special color lists is a 'wheel_item', described below.
- *WheelItem* : A wheel Item is a dataclass, which handles one named collection of colors. It contains a list of *ColorItem* objects and gives them a label. This is the base of one color definition.

1.2.3 YAML Definition

The easiest way to define colors is the code your own YAML file. We have a whole section dedicated to YAML, read more in *YAML Color Definitions*. You can also find an example color definition file in the examples directory.

1.2.4 Colors in code

You may want to add a color definitions to the pool of available colors (which is managed by *ColorwheelsConfig*). As soon as the color definition you create is added, it will be available to all *Colorwheels* instances.

A color definition is basically a list of colors (defined by the *ColorItem* dataclass) with a name attached. This is bundled in the *WheelItem* dataclass.

So, adding your own RGB sequence using code could look like this:

```
# mywheels.py
#
# Colorwheel Generator Example 4

from typing import List
import colorwheels

def my_rgb():
    """Create a list of colors (rgb)"""
```

(continues on next page)

(continued from previous page)

```
    color_list = list()
    color_list.append(colorwheels.ColorItem(red=255, green=0, blue=0))
    color_list.append(colorwheels.ColorItem(red=0, green=255, blue=0))
    color_list.append(colorwheels.ColorItem(red=0, green=0, blue=255))

    return color_list

wheels = colorwheels.Colorwheels()
# add my new list named 'myrgb' to common configurations ->
# can be used by any other instance of Colorwheels
wheels.wheel_configurations.add_wheel_item(colorwheels.WheelItem("myrgb", my_rgb()))
```

1.2.5 Base Colors

Your application may depend on some base colors, not necessarily in a / any sequence. Colorwheels has you covered: you can invoke `add_base_colors` in *ColorwheelsConfig* and these colors will be available to you. The idea is, that the generator can serve only one color, again and again. In any format.

The `add_base_colors` method adds the following colors to your configuration file: 'red', 'green', 'blue', 'cyan', 'magenta', 'yellow', 'black', and 'white'.

We use this feature not only for static basic colors, but also in solutions, where a background and foreground colorwheel are needed. For the foreground, we use a rotating Colorwheel based on the aesthetics of the solution, but for the background we start we a static color, say black. Now if we want to rotate both the foreground and background, we simply activate a different background. Neat hallucinogenous effects can be achieved, on the same color engine.

Base colors can be enforced by using the `add_base_colors` flag in `load_wheels` of *ColorwheelsConfig*, or when creating a new *Colorwheels* instance. In both cases, the default is set to `True`, so you'll probably end-up having them in your color collection.

Tip: you may want to over-ride a base color setting for one reason or another. For example, if you want your specific flavor of 'red' instead of the default '(255,0,0)', simply define it in your YAML configuration file; `add_base_colors` adds colors by name, only if they don't exist yet.

1.3 YAML Color Definitions

1.3.1 Introduction

Defining a color sequence in code is tedious. Definining more sequences even more so. Our package depends on color definitions coming from a YAML file.

This page describes the structure of such a file.

1.3.2 Specification

One or more color definitions are located in a YAML file, defining color sequences used in your program.

YAML structure

The base YAML structure is as follows:

```yaml
meta:
    release: "0.5.0.0"
wheels:
    - wheel:
        name: "white"
        colors:
            - rgb: [255, 255, 255]
    - wheel:
        name: "RGB"
        colors:
            - rgb: [255, 0, 0]
            - rgb: [0, 255, 0]
            - rgb: [0, 0, 255]
    ...
```

The structure contains 2 segments:

- **meta** - containing metadata about the file, version (for compatibility reasons) and similar. In the current release, this section is ignored.
- **wheels** - a list of different (named) colorwheels. Each colorwheel item is defined within a *wheel* section.

Wheel Definitions

Wheels can be represented in several ways, depending on the color effect we want to achieve. Generally, we have 2 types of *wheel* elements: listed, or calculated.

For example:

```yaml
...
- wheel:
    name: "long-rainbow"
    type: "rainbow"
    size: 64
    amplitude: 127
    center: 128
    frequency: 0.3
- wheel:
    name: "RGB"
    type: "sequence" # default value, need not be specified
    colors:
        - rgb: [255, 0, 0]
        - rgb: [0, 255, 0]
        - rgb: [0, 0, 255]
...
```

Above, we can see a standard RGB sequence, plus a 'calculated' sequence from parameters. The differentiator is the **type** field. This field identifies what to do with parameters, and defaults to **sequence**.

If you load the above definition into `Colorwheels` (using the `ColorwheelsConfig` `load_wheels` method), there will be 2 named color wheels you can switch back and forth.

Wheel types

sequence

Sequence is the basic type of a wheel, and if not specified, this type becomes the default value. A 'sequence' contains a 'colors' list, where every list item is defined by it's RGB elements.

Below a usage example:

```
...
- wheel:
    name: "RGB"
    type: "sequence" # default value, need not be specified
    colors:
        - rgb: [255, 0, 0]
        - rgb: [0, 255, 0]
        - rgb: [0, 0, 255]
...
```

rainbow

Rainbow is a generated wheel. You specify how many elements to use, plus some algorithm parameters. You can generate rainbow sequences without really looking into the detail of the implementation: just specify 'size' (i.e. number of colors) and leave the rest to defaults.

Below a rainbow wheel:

```
...
- wheel:
    name: "my-rainbow"
    type: "rainbow"
    size: 32        # default value, need not be specified
    amplitude: 127  # default value, need not be specified
    center: 128     # default value, need not be specified
    frequency: 0.3  # default value, need not be specified
...
```

1.4 Colorwheels

1.4.1 Introduction

A Color management module, containing the `Colorwheel` generator. As described in the code documentation, the idea of the colorwheel is to choose a sequence of colors, and, by using a `next` call, select the next color in the chain. Then start at the beginning again.

Our generator allows to multiple return types. You can tweak the return values by setting a `generator_type` to any one of those values: `"rgb_tuple"`, `"rgba_tuple"`, or `"hexadecimal"`. If calling standard `next` methods, 3 are provided for each mentioned type of return value.

1.4.2 Specification

`Colorwheels` is a module to generate color sequences. A Colorwheels instance contains multiple color wheel definitions and functions as a generator of color values in multiple formats. You can activate any available colorwheel anytime, to obtain different effects.

class colorwheels.colorwheels.**Colorwheels**(*add_base_colors=True*)

Base class for returning color sequences.

We represent colors similar to Color Wheels in photography, i.e. a sequence of colors is located on an imaginary wheel and endlessly served to applications in a given sequence.

Colorwheels can be generated or loaded from a YAML file. See documentation and examples.

__init__(*add_base_colors=True*)

Create `Colorwheels` instance.

wheel configurations is a *ColorwheelsConfig* object (singleton), which contains color definitions loaded from the environment, or generated by code.

A default configuration object is created (or inherited from other parts of the code).

activate_colorwheel(*name*)

Activates colorwheel by name, from configuration file. Sets active_wheel with new setting.

Raises **ValueError** – Raises ValueError exception if name is not found

property active_colors

Color list of active colorwheel.

Exposes the active colorwheels color list for easy iteration

property active_name

Currently active colorwheel name

complement()

Use own colors to create a compementing color palette.

Current colors are overwritten.

Tip: to synchronize two colorwheels, create complements before using the generator(s)

See also:

rainbow You can also change the generator colors by creating a rainbow effect

next()

Get the next color from the ColorWheel as an RGB tuple

Returns RGB tuple of next selected color. The returned value is an integer tuple representing a color, i.e. (r,g,b). Red would return (255,0,0)

Return type tuple

See also:

next_rgba Get the next color from ColorWheel using RGBA

next_hex Get the next color from ColorWheel as a hex string

`next_hex()`
Get the next color from ColorWheel as a hex string

Returns hex string representation of RGB color

Return type string

See also:

next Get the next color from the ColorWheel as an RGB tuple

next_rgba Get the next color from ColorWheel using RGBA

`next_rgba(alpha=255)`
Get the next color from ColorWheel using RGBA

Returns RGB tuple of next selected color. The returned value is an integer tuple representing a color, i.e. (r,g,b,a). Red would return (255,0,0,255) if alpha is default at 255

Return type tuple

See also:

next Get the next color from the ColorWheel as an RGB tuple

next_hex Get the next color from ColorWheel as a hex string

`rainbow(size)`
Generate rainbow color palette, using defaults.

Current colors are overwritten. A new, suitable name is generated.

Tip: to generate rainbow effects with more options, use the related rainbow class method from wheel_item

Parameters `size` – number of rainbow colors to be generated

See also:

complement You can also change the generator colors by replacing current colors with complementing colors

`set_generator_type(new_type)`
Set the generator type to a value out of generator_types.

Raises `ValueError` – Raises ValueError exception if new_type is not avaialble

`property wheel_configurations`
Returns the configuration object *ColorwheelsConfig*, used by Colorwheels

You can manage the configurations from any Colorwheels instance used in your program. Handle with care! The configurator is a singleton, i.e. if you for example load a different set of colors, all running generators will be affected.

1.5 ColorwheelsConfig

1.5.1 Introduction

ColorWheelConfig is a configuration helper for *Colorwheels*

1.5.2 Specification

ColorwheelsConfig is a configuration helper for *Colorwheels*, implemented as a singleton.

The helper loads a configuration YAML file and serves the colors by name to a Colorwheels generator.

`class colorwheels.colorwheels_config.ColorwheelsConfig(*args, **kwargs)`
Configuration helper for *Colorwheels*.

`__init__()`
Initialize configuration helper for *Colorwheels*.

The constructor creates the most simple configuration, and we expect a configuration file to be loaded later by code.

The initial wheel available is a primitive with 'red', 'green', 'blue', under the wheel name 'default'. This is equivalent to an RGB definition.

`add_base_colors()`
This method adds base colors to the list of available colors in colorwheels, to ensure availability of often used colors.

These simple color sequences have only one color available, so running the generator returns the same color all over again. It can come in handy for example, if you have 2 colorwheels for foreground/background, and want to have the background defaulting to black only ....

The method adds the following one-color named wheels:

'red', 'green', 'blue', 'cyan', 'magenta', 'yellow', 'black', 'white'

`add_wheel_item(item)`
Adds a *WheelItem* to definitions list.

Parameters `item` – A *WheelItem* object, which should be added to global configurations

Raises `ValueError` – Raises error if item name already exists

`create_wheel_item(name, colors)`
Create a *WheelItem* from parts. Function returns the created item

Parameters

- `name` – Name your new *WheelItem*
- `colors` – Supply a list of of *ColorItem* color objects

`find_wheel(name)`
Find wheelitem by name. None if not found

`property first_wheel`
Get the first wheel available

`load_wheels(filename, add_base_colors=True)`
loads YAML color definition file. The loaded file is converted to a list of *WheelItem* objects.

When loading a new definition from YAML, make sure colorwheels activates whatever wheel is required!

Parameters **filename** (*filename of file containing color definitions in YAML format. See*) – *YAML Color Definitions* for more details

Raises

- **FileNotFoundError:** – If file is not found on system
- **YAMLError:** – If file is a wrongly formatted YAML file

property wheel_names
Return list of wheel names available in configuration

1.6 ColorItem

1.6.1 Introduction

We keep color information and color logic together. **ColorItem** is a dataclass to simplify handling of color elements (RGB and others).

The class keeps native information about the RGB parts in dedicated fields. It can represent a color in one of the formats listed below.

- *RGB - ColorItem* instance: this is the native representation of a color
- *RGB - tuple*: You can retrieve color information as an RGB tuple. A red would become *(255, 0, 0)*
- *RGB - normalized tuple*: a color in some systems has to be represented by float values from 0 to 1. The normalized tuple can represent this. The red example would look as follows: *(1.0, 0.0, 0.0)*
- *RGBA - tuple*: PIL and others sometimes work with RGBA tuples. The color information is enriched with the Alpha information, in this release hardcoded as '255'. Our red would be represented as *(255, 0, 0, 255)*
- *RGB hexadecimal*: the color can be sent as an RGB hexadecimal string.

A few conversion color methods are bundled together with color information.

1.6.2 Specification

ColorItem is a dataclass containing one Color definition and color converters.

The object contains the following values:

- red: red component of color (integer)
- green: green component of color (integer)
- blue: blue component of color (integer)

Further, color format conversions - in between RGB, RGBA (integer representation) and float representations - are provided.

class colorwheels.color_item.**ColorItem**(*red: int*, *green: int*, *blue: int*)
Color object (dataclass) for easy color handling.

Parameters

- **red** (*int*) – red color component. The native format is an integer 0-255
- **green** (*int*) – green color component. The native format is an integer 0-255
- **blue** (*int*) – blue color component. The native format is an integer 0-255

property color
The *color* property return color as tuple.

Returns A tuple of 3 elements, red, green, blue. A red is returned as *(255, 0, 0)*

Return type tuple

property color_hex
Return hexadecimal representation of color.

Returns Hexadecimal coded string. A red is returned as *#ff0000*

Return type str

property complement
Return complement (opposite) color tuple.

Returns Finds a complementing color to current, and returns it as a tuple. Our example red color - *(255, 0, 0)* is complemented by *(0, 255, 255)* - cyan.

Return type tuple

from_float(*colrgb*)
Convert a float RGB tuple the native format (int tuple)

This method comes in handy, if you use libraries like 'Colour' in your code.

The Colour library uses RGB float values, encoded in tuples ranging from 0.0-1.0. We convert these values to an int tuple, i.e. int values ranging from 0-255

1.7 WheelItem

1.7.1 Introduction

WheelItem is a dataclass to simplify handling of colorwheel definitions. The structure of data closely mirrors YAML definition file(s) used to define colors.

A Wheel Item simply contains a named RGB color sequence.

Naming the sequence is used for easy switching of colorwheels, while the list of colors is a list of ColorItem objects. See *ColorItem* for more information.

1.7.2 Specification

WheelItem: a dataclass containing one ColorWheel definition

The object contains the following values:

- name: ColorWheel name. The name is used to retrieve a named color sequence
- colors: List of ColorItem colors

WheelItem encapsulates a colorwheel selection and is used internally by colorwheels.

class colorwheels.wheel_item.**WheelItem**(*name: str*, *colors: List[*colorwheels.color_item.ColorItem*]*)
Content of one colorwheel

classmethod complement_wheel_item(*reference_wheel*, *name=''*)
Use the reference wheel to create a similar, but color complementing wheel item.

If no name is provided, uses original name with the '_complement' suffix

from_float_list(*color_list*)
Convert a list of float RGB tuples to native format

This method comes in handy, if you use libraries like 'Colour' in your code.

The Colour library uses RGB float values, encoded in tuples ranging from 0.0-1.0. We convert these values to an int tuple, i.e. int values ranging from 0-255

Parameters **color_list** – A list of colors in float format(0.0-1.0 for each segment)

generate_rainbow(*size*, *amplitude*, *center*, *frequency*)
Generate colors with a Rainbow palette. Overwrites colors list.

Uses a simplified algorithm.

Tip: If you don't wish to experiment with the algorithm, you can decide on the rainbow size (number of colors), and use these values as a starting point:

amplitude=127, center=128, frequency=0.3

property is_single_color
Indicates, if color list contains only one color

classmethod rainbow_wheel_item(*name*, *size*, *amplitude=127*, *center=128*, *frequency=0.3*)
Generate a wheel item with a Rainbow palette. Provides some sensible defaults.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

C

INDEX

Symbols

A

C

F

G

I

L

M

N

R

S

W