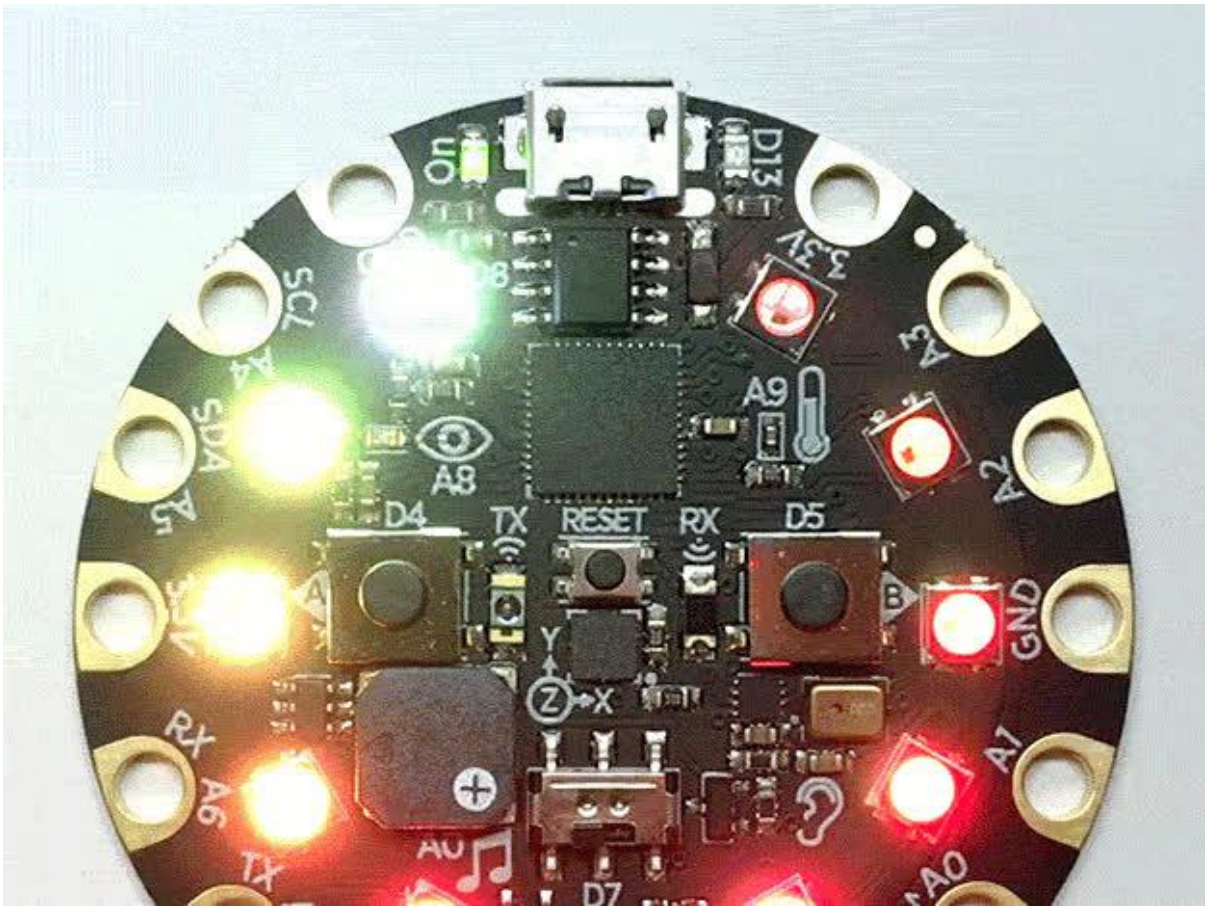
FancyLED Library for CircuitPython

Created by Phillip Burgess



https://learn.adafruit.com/fancyled-library-for-circuitpython

Last updated on 2021-11-15 07:06:43 PM EST

Table of Contents

Overview

FancyLED is a CircuitPython library to assist in creating buttery smooth LED animation. It's loosely inspired by the FastLED library for Arduino (https://adafru.it/eip), and in fact we have a "helper" library using similar function names to assist with porting of existing Arduino FastLED projects to CircuitPython.

Things to Know about FancyLED (vs FastLED):

- FancyLED does not "speak" any LED protocols on its own; it needs to work in conjunction with another library that handles the device specifics, such as NeoPixels or DotStars.
- FancyLED implements only a subset of FastLED features; those we had an immediate need for. This might expand over time, but will probably never be fully equivalent.
- FancyLED is not especially fast. Whereas FastLED relies on a lot of bit-level numerical tricks for performance, we don't really have that luxury in Python. Some design choices were made with an eye to the future, when more microcontrollers will have floating-point math capability, so certain operations will be a bit pokey on current-day hardware.
- FancyLED has different function names and arguments. With the helper library, function names are kept the same as FastLED where possible, though they don't always follow preferred Python style, and the arguments and return values may be changed somewhat. Keeping it as close as we can though.

Basically, if you need really hardcore performance LED stuff, and are comfortable in the Arduino environment, stick with FastLED! If you want something similar for CircuitPython, FancyLED is a start.

Install Software

Install & Use

You'll find adafruit_fancyled in the Adafruit CircuitPython Library Bundle, downloadable here if you don't already have it:

Click for the Latest Adafruit CircuitPython Library Bundle Release

https://adafru.it/Ayy

Place the `adafruit_fancyled` folder in the "lib" folder on your USB-connected CircuitPython device. CircuitPython library installation is covered in more detail in the Welcome to CircuitPython guide (https://adafru.it/ABU).

At the start of your CircuitPython code, import the adafruit_fancyled library...you can "import as" and assign it a shorter name that's easier to type, such as "fancy":

```
import adafruit_fancyled.adafruit_fancyled as fancy
```

To work with NeoPixel strips on most pins, also import the "board" and "neopixel" libraries:

```
import board
import neopixel
```

For the built-in NeoPixels on Circuit Playground Express, `import cpx` from the `adafruit_circuitplayground.express` library (NeoPixel support is built-in as part of cpx):

```
from adafruit_circuitplayground.express import cpx
```

There are two example scripts, cpx_rotate.py (for Circuit Playground Express) and neopixel_rotate.py (for NeoPixels on any digital pin) that demonstrate the library in use.

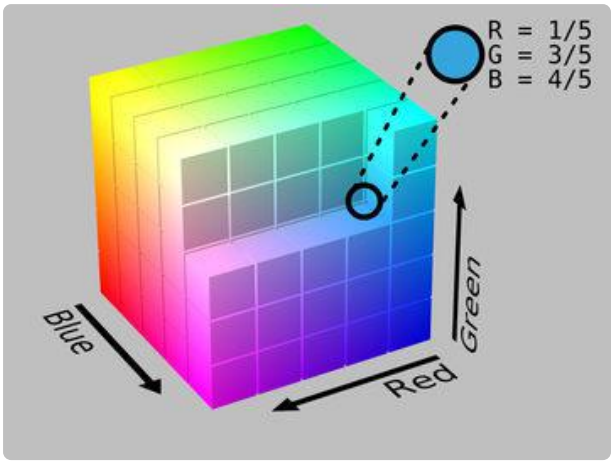
Colors

A few data types in FancyLED are used for specifying colors.

In these examples, we'll assume you've imported the adafruit_fancyled library as "fancy":

```
import adafruit_fancyled.adafruit_fancyled as fancy
```

RGB Colors



Anyone doing graphics or LED work is likely familiar with the RGB (red, green, blue) color space. In FancyLED these three components are encapsulated in the CRGB class to be passed around as a single entity. The color space in FancyLED is “normalized” — that is, values range from 0.0 to 1.0 (floating-point) rather than the 0 to 255 (integer) usually seen elsewhere. This reduces quantization artifacts when multiple operations are performed on colors.

Image credit: By RGB_farbwuerfel.jpg: Horst Frank RGB_color_solid_cube.png: SharkD derivative work: SharkD Talk [CC BY-SA 3.0 (https://adafru.it/Ay2)], via Wikimedia Commons

The CRGB constructor accepts three arguments — red, green and blue in that order:

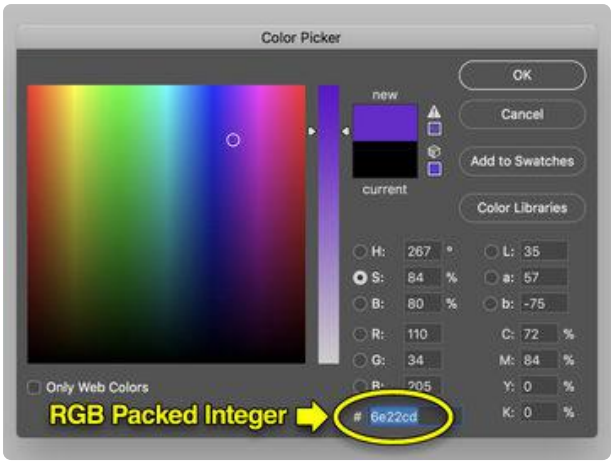
```
color = fancy.CRGB(1.0, 0.3, 0.0)  # Orange
```

Values outside the 0.0 to 1.0 range will be clipped to those bounds. However, if you prefer oldschool “0 to 255” colors you can also call the constructor using integers:

```
color = fancy.CRGB(255, 85, 0)  # Orange
```

Integer values are immediately converted to the normalized floating-point range on initialization, which can be seen by examining the CRGB instance’s red, green and bl ue attributes:

```
&gt;&gt;&gt; print(color.red, color.green, color.blue)
1.0, 0.333333, 0.0
```



In certain circumstances you might want or need to express an RGB color as a packed integer, usually in hexadecimal format. You'll see this format sometimes in HTML documents or the Photoshop color picker.

The first pair of hexadecimal digits are the red component (00 to FF), second pair is green, third is blue.

FancyLED can handle RGB colors in this format...

```
color = 0x6E22CD  # Purple
```

...but be aware that most of the library's functions will dissect this value and hand you back a CRGB type with the separate floating-point red, green and blue attributes. CRGB is "more native" to the library.

If you need something converted back into a packed integer value, this can be done through the library's pack() function:

```
packed = color.pack()
```

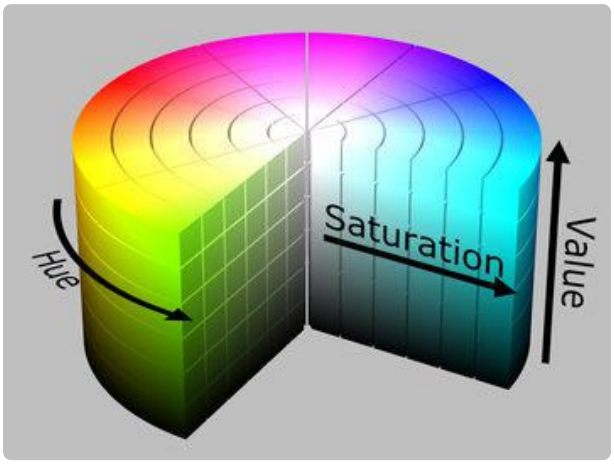
There's an inverse function, unpack(), for converting RGB packed integers to CRGB colors:

```
color = fancy.unpack(0x6E22CD)  # Purple
```

RGB packed integers are more space-efficient than the CRGB type — say if RAM is a concern and your code has a big list of colors. But integer colors may suffer quantizat ion artifacts after repeated operations...it's not good to go back and forth a lot.

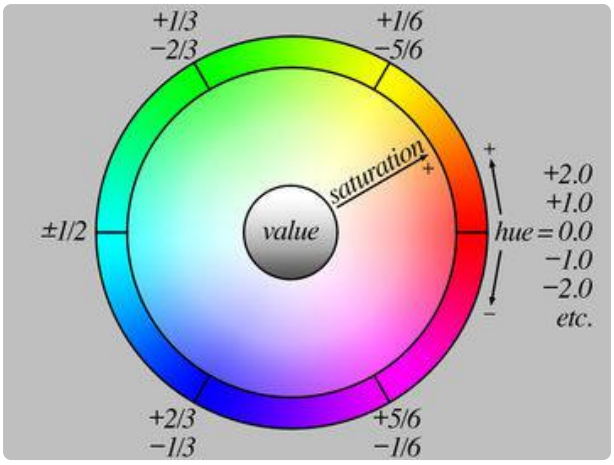
One place packed RGB values are used is with a corresponding LED-driving library (e.g. NeoPixel, DotStar). Since these libraries don't recognize CRGB colors natively, you can `pack()` a color into an integer before assigning it to an LED through the appropriate library. Remember that FancyLED doesn't actually talk to LEDs itself, it just handles color operations that are common to LED projects.

HSV Colors



Some folks are more comfortable thinking in the HSV (hue, saturation, value) colorspace, where colors are expressed as a position around the color wheel, how saturated they are away from gray, and their brightness. In FancyLED these are encapsulated in the CHSV class. Again these values are "normalized" — in the 0.0 to 1.0 range — though hue is a special case...

Image credit: By RGB_farbwuerfel.jpg: Horst Frank RGB_color_solid_cube.png: SharkD derivative work: SharkD Talk (RGB_farbwuerfel.jpg RGB_color_solid_cube.png) [CC BY-SA 3.0 (https://adafru.it/Ay2)], via Wikimedia Commons



Hue isn't constrained...it can be any value, but it "wraps around" at zero such that fractional values always represent the same hue...for example, -0.8, +0.2 and +1.2 are all the same green-tinged yellow.

Hue 0 is red. Yellow is +1/6, green is 2/6 (or 1/3), cyan is 3/6 (1/2) and so forth.

The `CHSV` constructor accepts three arguments — hue, saturation and value in that order:

```
color = fancy.CHSV(0.08, 1.0, 1.0)  # Orange
```

0–255 integers are also permitted, but like the CRGB constructor these are converted to floating-point internally:

```
color = fancy.CHSV(20, 255, 255)  # Orange
```

It's very common to just want a pure hue (no saturation or brightness adjustment), so it's okay to pass just a single value; the others will be set to 1.0 defaults:

```
color = fancy.CHSV(0.08)  # Orange
```

You can see this in action by then inspecting the CHSV instance's hue, saturation and value attributes:

```
&gt;&gt;&gt; print(color.hue, color.saturation, color.value)
0.08, 1.0, 1.0
```

Types, Conversions and Other Operations

While many FancyLED functions can operate on CHSV colors, some will return a CRGB type. This is normal and by design...some operations just don't translate directly to HSV colorspace, so the color is converted to RGB first.

FancyLED has no "packed integer" equivalent for HSV colors; that's an inherently RGB-oriented representation. If you call `pack()` with a CHSV color, it will be converted to RGB and the return value will be a packed integer representation of the RGB value.

You can manually convert an CHSV color to CRGB by passing the former to the latter's constructor:

```
color1 = fancy.CHSV(0.08, 1.0, 1.0)
color2 = fancy.CRGB(color1)
```

There's currently no inverse (RGB to HSV) function...if entropy has a direction in FancyLED, it's toward RGB. Perhaps later, with some decisions made. While HSV to RGB conversion always yields one unique answer, the inverse is sometimes ambiguous — there could be multiple solutions.

You can blend (interpolate) between two colors using the `mix()` function. This accepts two colors of any supported type (CRGB, CHSV or packed RGB integer...they don't need to be the same) along with a "weight" of the second color in the mix, in the range 0.0 to 1.0...e.g. 0.0 is 100% the first color, 1.0 is 100% the second color, and 0.5 is an equal mix between the two colors.

```
color1 = fancy.CHSV(0.08, 1.0, 1.0)
color2 = fancy.CRGB(255, 85, 0)
color3 = fancy.mix(color1, color2, 0.25)  # 75% color1, 25% color2
```

If both colors are CHSV type, interpolation takes place in HSV space. This is important for giving a "direction" for the hue change, and dealing with ranges that "cross the seam" at hue 0.0. For example, blending from cyan to red...do you want the midpoint color to be green or purple? How you choose your endpoint hues will determine this (0.5 to 0.0 will be greens and yellows along the way...0.5 to 1.0 will be blues and magentas...but the ends will be the same). The return value in this case will also be a CHSV type.

If one or both colors are CRGB type or packed integer, interpolation takes place in RGB space (CHSV color, if any, is converted to RGB first) and the return value is a CRGB type.

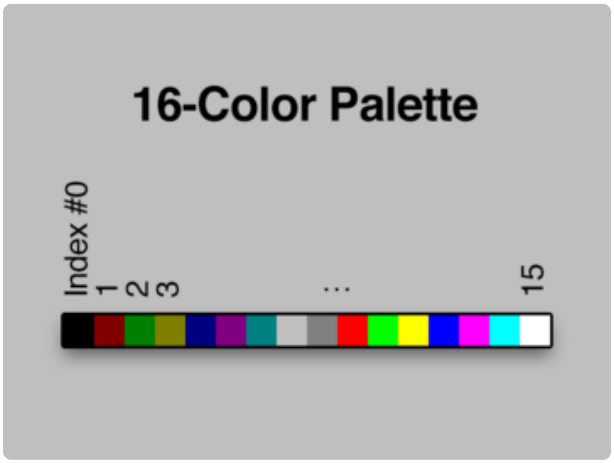
`mix()` can be used to create a color gradient to be issued to LEDs, or to fill a list...

```
color1 = fancy.CRGB(1.0, 0, 0)    # Red
color2 = fancy.CRGB(1.0, 1.0, 0)  # Yellow
colorlist = [fancy.mix(color1, color2, i / 9) for i in range(10)]
```

Palettes

A palette is a collection of colors that you want to use to make special lighting effects. By pre-choosing your colors you can play around with effects and keep within your "theme."

For example, if you want to have a fire/flame looking project, pick a palette with red, orange, yellow and maybe white. If you want an ice/frost looking project, pick a palette with dark blue, cyan, and white. Nature lovers will want green tints.

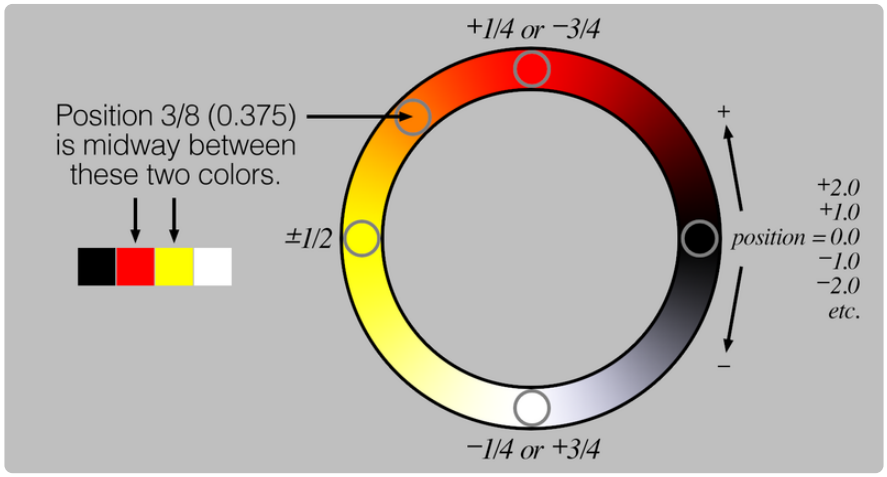


Traditionally in computer graphics, color palettes have a start and an end, and a finite number of entries, usually a power of two, often dictated by graphics hardware. 16- and 256-entry color palettes are commonly seen, and that's what the FastLED Arduino library supports.

FancyLED is just software…it's not bound by physical hardware constraints…and its concept of color palettes is a little different. The number of palette entries can be any thing…could be five, could be 50. The sequence "loops around" — there doesn't need to be a start or finish, and you can reference colors between the palette entries, it's not just discrete steps.

Like hue explained on the prior page, colors are drawn from a palette with a floating-point index, where 0.0 and 1.0 represent the beginning of the list and…well, not the end, but back to the beginning of the list. It can be any number but will "wrap around" in this range. This is a boon for LED animation, which tends to be cyclical.

Here's how a color palette with four entries might be referenced:



Code-wise, color palettes in FancyLED are simply Python lists where each entry is a CRGB, CHSV or packed integer color:

```
my_palette = [fancy.CRGB(0, 0, 0),       # Black
              fancy.CHSV(1.0),           # Red
              fancy.CRGB(1.0, 1.0, 0.0), # Yellow
              0xFFFFFF]                  # White
```

You can mix-and-match color types if you like…they don't need to all be CRGB or CHSV. A palette can also be a tuple (bounded by (parenthesis) rather than [brackets]) if you know your palette is set in stone (or want to enforce that). Lists give more flexibility for replacing or adding new elements.

Use the `palette_lookup()` function to retrieve a color from a palette list, passing the palette name and floating-point position as arguments:

```
color = fancy.palette_lookup(my_palette, 0.3)
```

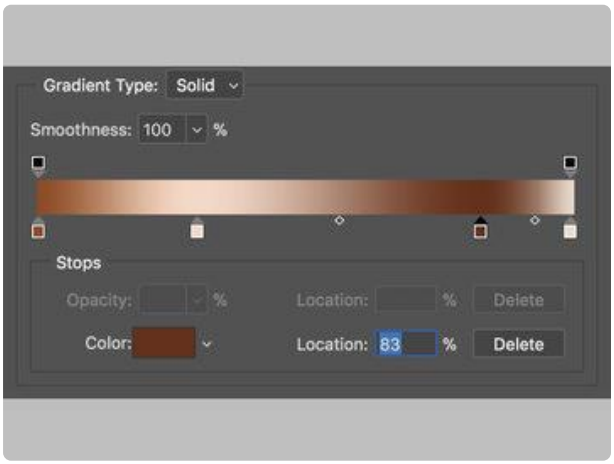
Like the `mix()` function explained on the previous page...if the requested color is between two CHSV colors, the return value of `palette_lookup()` will also be a CH SV type. In all other cases it returns a CRGB type (even if the palette is using packed integers).

The circular nature of FancyLED palettes does not preclude access to the original discrete colors, or treating the last entry as the palette end. One option is to use fractions in a call to `palette_lookup()`, where the denominator is the number of palette entries and the numerator is the index (0 to length-1) ... e.g. 0, 1/4, 2/4 and 3/4 with a 4-element palette like the one above. But in most cases it's easier to bypass the library and pull from the palette list directly:

```
color = my_palette[3]
```

Gradient Palettes

Taking a cue from FastLED and from Photoshop's gradient tool, there's a second type of color palette that's sometimes more concise, but requires some extra steps to set up...



Here's a view of Photoshop's gradient designer tool. Notice there are four "stops" (the markers just below the gradient)...but, unlike our color palettes described above, these are not evenly spaced. Each stop can be positioned anywhere along the gradient, though they do need to be in increasing positions along its length.

This one has stops at 0%, 30%, 83% and 100%.

A similar gradient in FancyLED would be expressed as a list of tuples, each containing two elements: the location of the stop (from 0.0 at the beginning to 1.0 at the end) and a color (any supported format — CRGB, CHSV or RGB packed integer).

The copper gradient above could be expressed as:

```
grad = [(0.0, 0x97461A),
        (0.3, 0xFBD8C5),
        (0.83, 0x6C2E16),
        (1.0, 0xEFDBCD)]
```

However...you can't just pull a color from a gradient list using `palette_lookup()`. The uneven spacing would make it inordinately slow.

Instead, we first convert the gradient to a regular (equal interval) palette using the `expand_gradient()` function, with two arguments: the gradient list source, and the number of entries we want in the resulting palette:

```
palette = fancy.expand_gradient(grad, 50)
```

How many entries in the output palette is a balance been RAM (always in short supply) and how faithfully we want to reproduce the gradient palette. The copper gradient above might look good enough with just a dozen palette entries (then interpolating in-between with `palette_lookup()`), while really intricate gradient setups might need 100 or more.

Once converted, colors can be read (from the converted palette, not the original) using `palette_lookup()`:

```
color = fancy.palette_lookup(palette, 0.42)
```

Assigning Colors to LEDs

So we can store and retrieve colors, and mix between them, but how to we actually get those colors to the LEDs?

FancyLED doesn't speak any addressable LED protocols on its own. We import an existing library (like NeoPixel or DotStar), then assign colors (from FancyLED) to LEDs (using the corresponding LED library), usually in a loop of some sort.

As explained on the Install page, for NeoPixels we need to import the "board" and "neopixel" libraries:

```
import board
import neopixel
```

Then create a NeoPixel instance, passing the board pin number, number of LEDs and optionally `auto_write=False` so the LEDs only refresh when we say so:

```
strip = neopixel.NeoPixel(board.D1, 10, auto_write=False)
```

Or for the built-in NeoPixels on Circuit Playground Express:

```
from adafruit_circuitplayground.express import cpx
```

The NeoPixel and DotStar libraries can both accept RGB packed integers as inputs, so we'll use FancyLED's `pack()` function to hand off the data.

Here's a complete CircuitPython script for some FancyLED animation on Circuit Playground Express. It creates a color palette with 4 entries, then loops forever, moving data from the color palette to each NeoPixel, with a slight offset each time around to make the palette "spin":

```
from adafruit_circuitplayground.express import cpx
import adafruit_fancyled.adafruit_fancyled as fancy

cpx.pixels.auto_write = False  # Update only when we say
cpx.pixels.brightness = 0.25   # make less blinding

palette = [fancy.CRGB(255, 255, 255),  # White
           fancy.CRGB(255, 255, 0),    # Yellow
           fancy.CRGB(255, 0, 0),      # Red
           fancy.CRGB(0,0,0)]          # Black

offset = 0  # Position offset into palette to make it "spin"

while True:
    for i in range(10):
        color = fancy.palette_lookup(palette, offset + i / 9)
        cpx.pixels[i] = color.pack()
    cpx.pixels.show()

    offset += 0.033  # Bigger number = faster spin
```

Secret Ingredient: Gamma Correction

There's an extra optional step we can perform if we want the LED colors to appear more "true" or if we'd like a global color balance adjustment...

Gamma correction is a process of adjusting brightness to better match the response curve of our eyes. Without it, mid-range levels appear unreasonably bright (50% brightness appears to our eyes more like 80%). We've written a whole guide (https://adafru.it/w2B) if you need more details.

FancyLED provides the `gamma_adjust()` function to perform this operation. This can operate on a single CRGB or CHSV color (but not RGB packed integer) or a whole list of colors — the first argument to this function, which is required, is the color to adjust, or a list of colors.

An optional named argument — `gamma_value` — is the exponent that's applied to the color(s)...either a single value (which is applied equally to red, green and blue), or a 3-element tuple of floats if you need separate exponents for red, green and blue. If nothing is specified, the default exponent is 2.7 which works reasonably well in most cases.

A second optional named argument — `brightness` — is a brightness adjustment that's applied in the same operation. Like `gamma_value`, this can be a single value (0.0 to 1.0) that's applied to red, green and blue, or a 3-element tuple of floats to adjust red, green and blue separately...pretty handy for adjusting the color balance on LEDs, which are seldom a neutral white. Default brightness is 1.0 (no dimming performed).

If converting a list of colors, one more optional named argument — `inplace` — decides whether a new gamma-adjusted color list is returned (keeping the original intact), or if the library may replace the contents of the list you passed in. By default this is `False` and a new list is returned.

Gamma adjustment should be the last step before colors are issued to the LEDs, and applied "on the way out." You probably don't want to keep the adjusted value around...repeated gamma adjustments over the same data will destroy your hard work!

Here's how we might add gamma and brightness adjustments to the example program above. First, after this line...

```
offset = 0  # Position offset into palette to make it "spin"
```

Add a second line, consisting of a 3-element tuple of floats. These are brightness levels for red, green and blue, respectively. Aside from bringing the LEDs down to a less eye-searing brightness level, this also tries to balance the colors better so white doesn't appear blue-tinged:

```
offset = 0  # Position offset into palette to make it "spin"
levels = (0.25, 0.3, 0.15)
```

Then, between these two lines in the pixel-setting loop:

```
color = fancy.palette_lookup(palette, offset + i / 9)
        cpx.pixels[i] = color.pack()
```

Insert a new line that processes “color” through the `gamma_adjust()` function, using our “levels” setting and putting the result back into the “color” variable:

```
color = fancy.palette_lookup(palette, offset + i / 9)
        color = fancy.gamma_adjust(color, brightness=levels)
        cpx.pixels[i] = color.pack()
```

Run the change and you’ll see the colors are a bit more mellow now.

Here are some different ways `gamma_adjust()` might be invoked, from the simple to the complex:

```
color = fancy.gamma_adjust(CRGB(0.6, 0.1, 0.9))

color1 = CHSV(0.5, 0.6, 0.3)
color2 = fancy.gamma_adjust(color1, gamma_value=2.7)

colorlist1 = [fancy.CRGB(128, 32, 117),
              fancy.CRGB(88, 85, 223),
              fancy.CRGB(0, 187, 30)]
levels = (0.25, 0.3, 0.15)
gammas = (2.6, 2.6, 2.7)
colorlist2 = fancy.gamma_adjust(colorlist1, gamma_value=gammas, brightness=levels)
```

Note that `gamma_adjust()` always returns a CRGB type (or list of CRGBs), even when the input(s) are CHSV. It’s an operation inherently based in RGB color space.

Whatever LED library you’re using — NeoPixel, DotStar or other — probably has its own brightness-setting function. Use one or the other, or the effect will compound and LEDs will be very dim. We recommend setting the NeoPixel/DotStar/other brightness to the maximum and handling this adjustment in FancyLED, since it provides color balance control and gamma correction.

FastLED Helper Functions

The fastled_helper library provides some “wrapper” functions around FancyLED that can simplify bringing over existing projects and data from FastLED.

This is imported separately and in addition to adafruit_fancyled:

```
import adafruit_fancyled.adafruit_fancyled as fancy
import adafruit_fancyled.fastled_helpers as helper
```

Some things don't require the helper functions at all, just some formatting changes to conform to Python syntax. For example, consider this 16-element RGB palette from one of the FastLED Arduino examples. Each palette entry is a single "packed" RGB value:

```
// Rainbow colors with alternatating stripes of black
#define RainbowStripesColors_p RainbowStripeColors_p
extern const TProgmemRGBPalette16 RainbowStripeColors_p FL_PROGMEM =
{
    0xFF0000, 0x000000, 0xAB5500, 0x000000,
    0xABAB00, 0x000000, 0x00FF00, 0x000000,
    0x00AB55, 0x000000, 0x0000FF, 0x000000,
    0x5500AB, 0x000000, 0xAB0055, 0x000000
};
```

In FancyLED for CircuitPython, the palette data itself can be kept exactly the same, but it's wrapped up in a Python list (surrounded in [square brackets]). None of the "PROGMEM" stuff is needed:

```
RainbowStripeColors = [
    0xFF0000, 0x000000, 0xAB5500, 0x000000,
    0xABAB00, 0x000000, 0x00FF00, 0x000000,
    0x00AB55, 0x000000, 0x0000FF, 0x000000,
    0x5500AB, 0x000000, 0xAB0055, 0x000000]
```

Meanwhile, other FastLED capabilities simply aren't present in FancyLED, even with helper functions. Color by name (such as "CRGB::DarkBlue" in FastLED) are not currently supported in FancyLED. Palettes must be specified numerically.

Gradient Palettes

Here's how a gradient palette would be specified in FastLED for Arduino, and it's converted to a "regular" palette (16 elements in this case) simply with an assignment:

```
DEFINE_GRADIENT_PALETTE( heatmap_gp ) {
    0,   0,  0,  0,   //black
  128, 255,  0,  0,   //red
  224, 255,255,  0,   //bright yellow
  255, 255,255,255 }; //full white

CRGBPalette16 myPal = heatmap_gp;
```

We can keep the list of numbers itself exactly the same with the FancyLED helpers, but the operations around it are changed.

First, the gradient palette data must be declared as a Python bytearray using the byte s() keyword...and the list itself is in both parenthesis and square brackets.

Second, we call loadDynamicGradientPalette(), passing the bytearray and desired palette length (in FastLED this is traditionally 16, 32 or 256, but in FancyLED can be any length, RAM permitting). This is slightly different from the same function in FastLED, where the second argument is a preallocated destination color palette rather than a list length):

```
heatmap_gp = bytes([
    0, 0, 0, 0,
    128, 255, 0, 0,
    224, 255, 255, 0,
    255, 255, 255, 255])
palette = helper.loadDynamicGradientPalette(heatmap_gp, 16)
```

Reading Colors From Palettes

Although the palette concept is similar between the two libraries, how things are indexed is a little different.

FastLED palettes typically have 16, 32 or 256 elements. But rather than a floating-point range, they use a fixed-point integer scale. Fetching “color 0” from a FastLED palette will return the first entry, 16 returns the second palette entry, 32 is the third and so forth. For values between these, an interpolated color between the two nearest palette entries can be returned…for example, an index of 8 will return a color about midway between the first and second palette entries. So, a 16-element palette will expect a range from 0 to 255 (values above 240 will “wrap around” to the first color when interpolating), 256-element palettes expect a range of 0 to 4095.

```
color = helper.ColorFromPalette(palette, index, brightness, blend)
```

First two arguments — palette and index — have been explained above.

Third argument is a brightness adjustment from 0 (minimum) to 255 (maximum). This is optional…the default value will be 255 (full brightness). Whatever LED library you’re using (e.g. NeoPixel) will usually have its own brightness setting, which is compounded (e.g. if 50% brightness in both libraries, the result will be 25% bright). If using FancyLED’s gamma controls, it’s best to handle brightness there and leave everything else at full brightness.

Fourth argument is a flag to enable (True) or disable (False) color blending for values between multiples of 16. Default is False…pass True to enable blending between palette indices.

Return value is a CRGB color, which can then be handled with other FancyLED functions.

Other Color Functions

```
color2 = applyGamma_video(color1, gamma_red, gamma_green, gamma_blue, inplace)
```

This is simply a wrapper around FancyLED's gamma_adjust() function to give it a similar syntax to FastLED. It function performs gamma correction (making brightness gradients appear more uniform) on a single brightness value, a single CRGB or CHSV color, or a list of CRGB or CHSV colors.

In the single-value case, only the value passed and the first gamma value are used; any other arguments are ignored. If no gamma value is specified, the default curve (2.5) will be used.

In the CRGB, CHSV, or list cases, separate red, green and blue gamma values can be specified. If only a single gamma value is supplied, this will be used across all three color components. If no gamma values are given, the default value (2.5) will be used.

Also in the CRGB/CHSV and list cases, the last argument — inplace — determines whether the value passed in will be modified "in place" (overwriting the original value(s) in that tuple or list) (True), or if the original data is left as-is and a new tuple or list (with gamma correction applied) is returned (False).

The color returned is always CRGB (or a list of CRGBs), even if the input was CHSV.

```
napplyGamma_video(n, gamma_red, gamma_green, gamma_blue)
```

Similar to the above function, but always modifies data in-place. For compatibility with FastLED code that uses the function of this name rather than the "inplace" flag.

```
hsv2rgb_spectrum(hue, saturation, value)
```

Given hue, saturation and value, returns an equivalent CRGB color. Roughly equivalent to the same function in FastLED.

Accepts three arguments: hue, saturation and value, each in the range 0 to 255.

What about all the cool noise and wave functions and stuff in FastLED?

Not yet, but we might add to this over time.