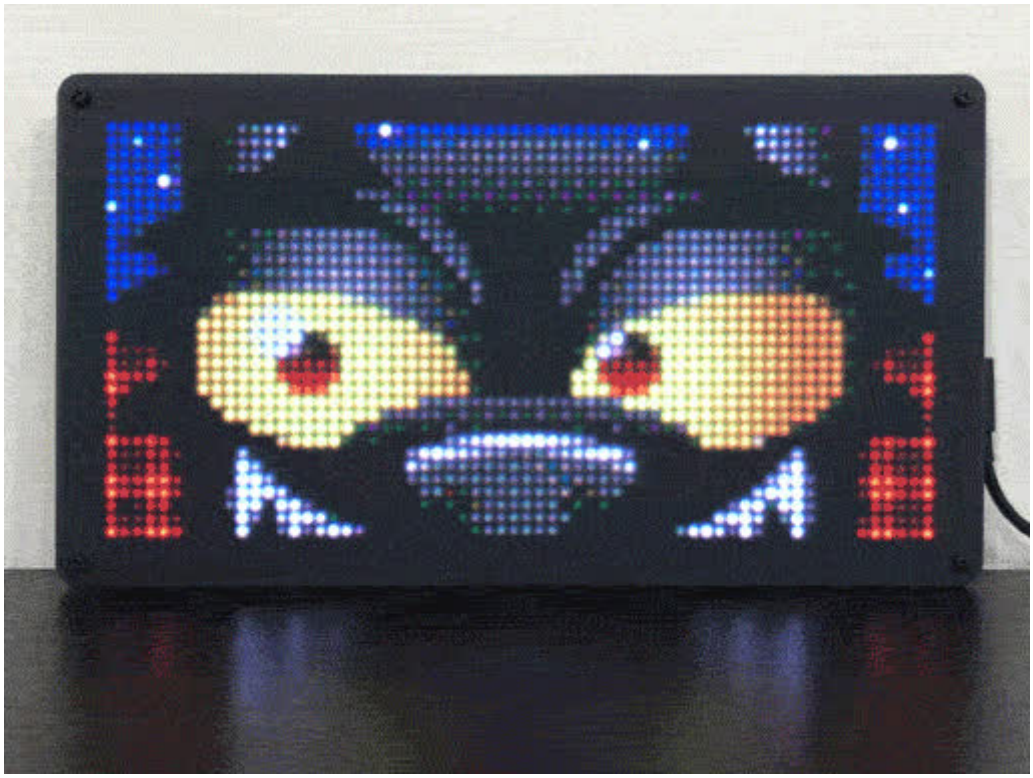
Matrix Portal Creature Eyes

Created by Phillip Burgess



https://learn.adafruit.com/matrix-portal-creature-eyes

Last updated on 2024-06-03 03:13:14 PM EDT

Table of Contents

Overview

Seems like every great **monster** film sets off a chain of **sequels**. Bride of..., Son of..., Revenge of..., Teen Wolf Too (hey, they can't all be winners).

Our **electronic eyeball** projects have spawned their own little franchise, with Pi Eyes (https://adafru.it/zrC), HalloWing (https://adafru.it/CEs), Monster M4SK (https://adafru.it/GfQ) and more (https://adafru.it/Ha4). The arrival of the **Matrix Portal M4** (http://adafru.it/4745) board made another sequel inevitable — this time a **retro-style** return to form of our first project of the series (https://adafru.it/iwB), but punching it up with bright colors, full-screen themes, and coded all in **CircuitPython** now.

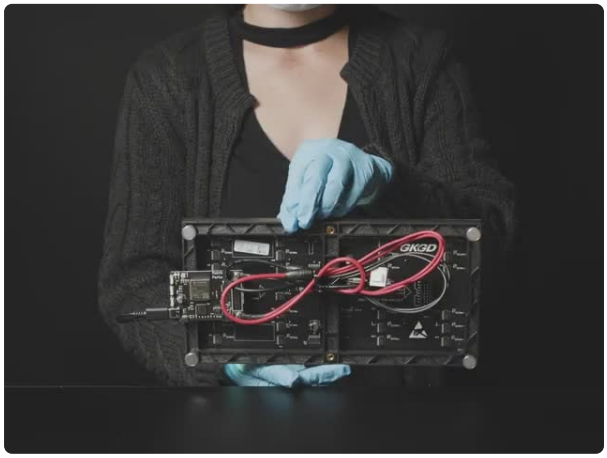
This makes a nifty **Halloween** window or tabletop display. And it's a not-too-daunting introduction to **CircuitPython** and **graphics**. When the Halloween season's over...if you don't keep the decorations up year 'round like some of us...everything can be repurposed into your own projects, or try out some others like a Moon phase clock (https://adafru.it/NB7).

This project requires:

- Adafruit **Matrix Portal M4** (http://adafru.it/4745) board
- Any of our **64x32** pixel "HUB75" (not NeoPixel) RGB LED matrices
- **USB C** cable
- USB **power supply** with output of 2 Amps or more

This guide will get the software running on the bare Matrix Portal hardware. Mounting or supporting the clock in an enclosure or frame is left as an exercise to the reader.

Parts

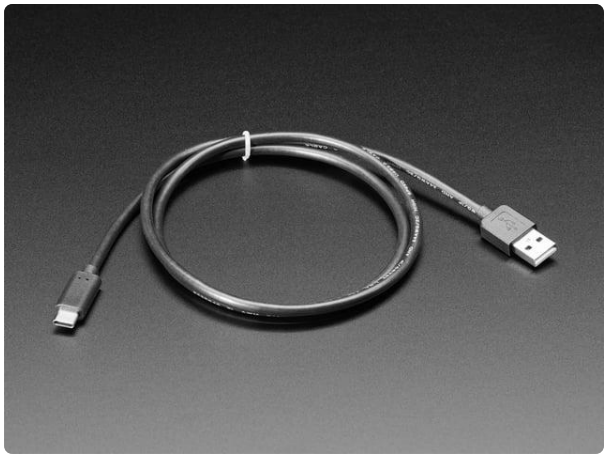


Adafruit Matrix Portal - CircuitPython Powered Internet Display

Folks love our wide selection of RGB matrices and accessories, for making custom colorful LED displays... and our RGB Matrix Shields...

https://www.adafruit.com/product/4745

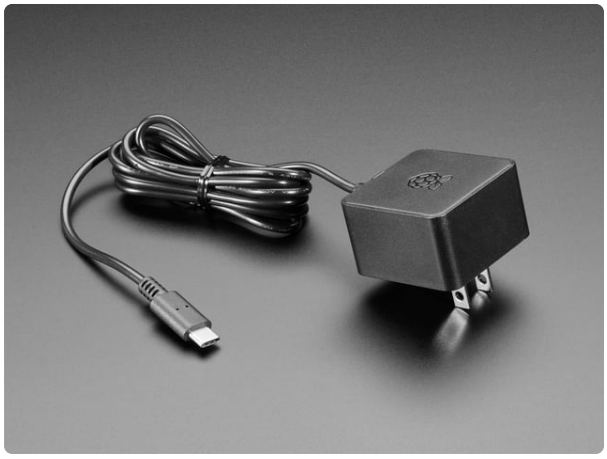
You can use a USB C power supply or a USB micro B with a micro B to C adapter (http://adafru.it/4299)



USB Type A to Type C Cable - approx 1 meter / 3 ft long

As technology changes and adapts, so does Adafruit. This USB Type A to Type C cable will help you with the transition to USB C, even if you're still...

https://www.adafruit.com/product/4474



Official Raspberry Pi Power Supply 5.1V 3A with USB C

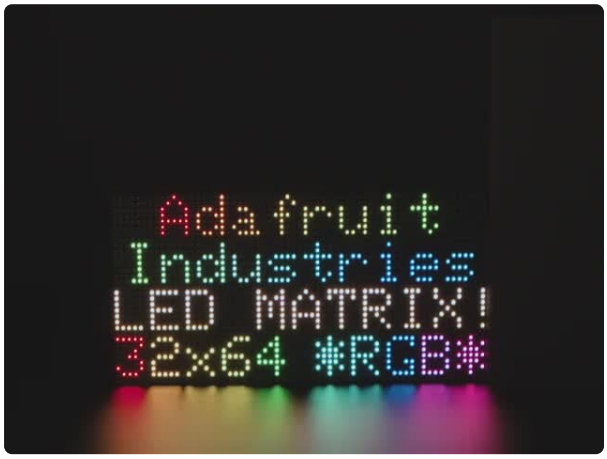
The official Raspberry Pi USB-C power supply is here! And of course, we have 'em in classic Adafruit black! Superfast with just the right amount of cable length to get your Pi 4...

https://www.adafruit.com/product/4298

[5V 2.5A Switching Power Supply with 20AWG MicroUSB Cable]
Our all-in-one 5V 2.5 Amp + MicroUSB cable power adapter is the perfect choice for powering single-board computers like Raspberry Pi, BeagleBone, or anything else that's...
https://www.adafruit.com/product/1995

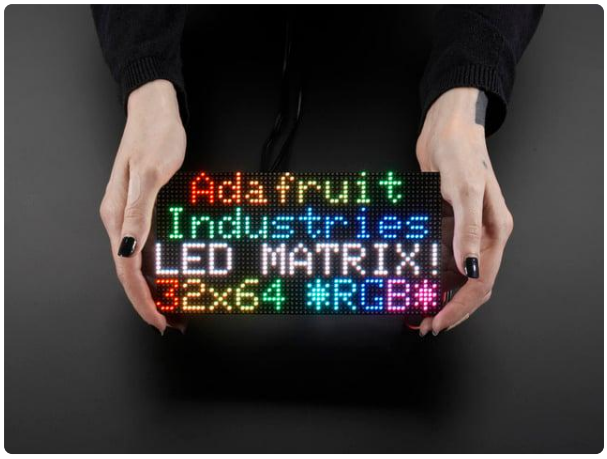
[Micro B USB to USB C Adapter]
As technology changes and adapts, so does Adafruit, and speaking of adapting, this adapter has a Micro B USB jack and a USB C...
https://www.adafruit.com/product/4299

If you'd like your LEDs diffused (and if your LED matrix is 4mm pitch or smaller), some acrylic may help:



[Black LED Diffusion Acrylic Panel 12" x 12" - 0.1" / 2.6mm thick]
A nice whoppin' slab of some lovely black acrylic to add some extra diffusion to your LED Matrix project. This material is 2.6mm (0.1") thick and is made of special cast...
https://www.adafruit.com/product/4594

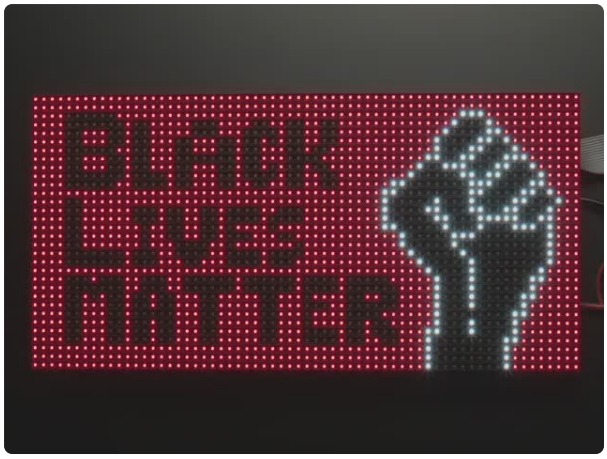
Adafruit carries a number of 64x32 RGB LED Matrices, varying between the space between LEDs (pitch) and whether rigid or flexible. Choose your favorite - larger pitch means the display is larger, width and height-wise but with the same number of pixels, and larger may be easier to read further away. Smaller for near your desk, for example.



64x32 RGB LED Matrix - 3mm pitch

Bring a little bit of Times Square into your home with this sweet 64 x 32 square RGB LED matrix panel. These panels are normally used to make video walls, here in New York we see them...

https://www.adafruit.com/product/2279



64x32 RGB LED Matrix - 4mm pitch

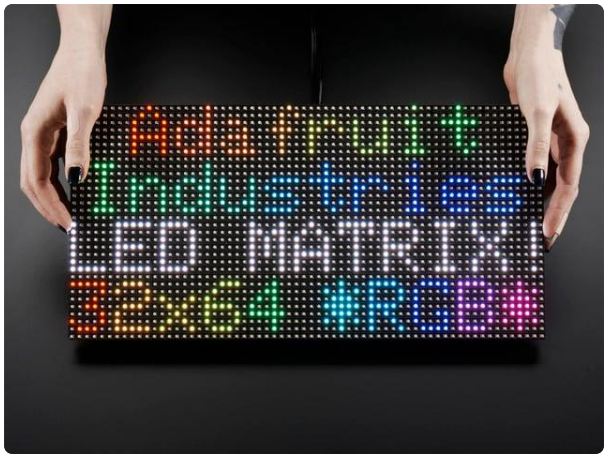
Bring a little bit of Times Square into your home with this sweet 64 x 32 square RGB LED matrix panel. These panels are normally used to make video walls, here in New York we see them...

https://www.adafruit.com/product/2278

64x32 RGB LED Matrix - 5mm pitch

Bring a little bit of Times Square into your home with this sweet 64x32 square RGB LED matrix panel. These panels are normally used to make video walls, here in New York we see them on...

https://www.adafruit.com/product/2277



64x32 RGB LED Matrix - 6mm pitch

Bring a little bit of Times Square into your home with this sweet 64x32 square RGB LED matrix panel. These panels are normally used to make video walls, here in New York we see them on...

https://www.adafruit.com/product/2276

64x32 Flexible RGB LED Matrix - 4mm Pitch

If you've played with multiplexed RGB matrices, you may have wondered "hey, could we possibly manufacture these on a thin enough PCB, so it's flexible?" and the...

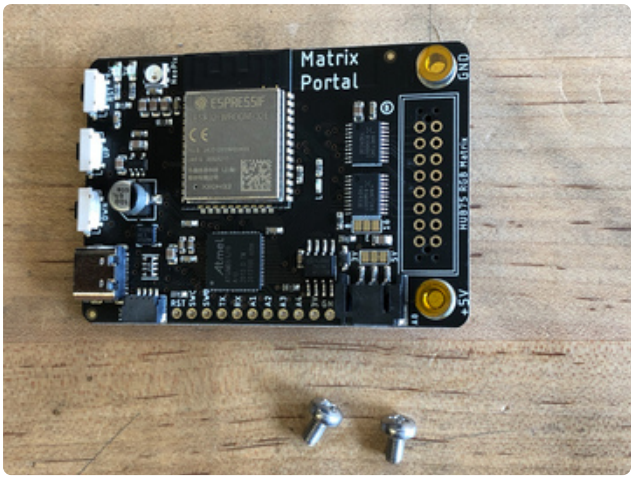
https://www.adafruit.com/product/3826

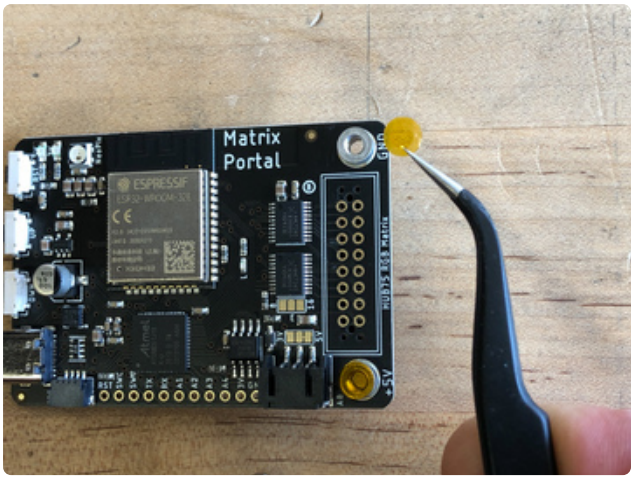
64x32 Flexible RGB LED Matrix - 5mm Pitch

If you've played with multiplexed RGB matrices, you may have wondered "hey, could we possibly manufacture these on a thin enough PCB so it's flexible?" and the answer...

https://www.adafruit.com/product/3803

Prep the MatrixPortal

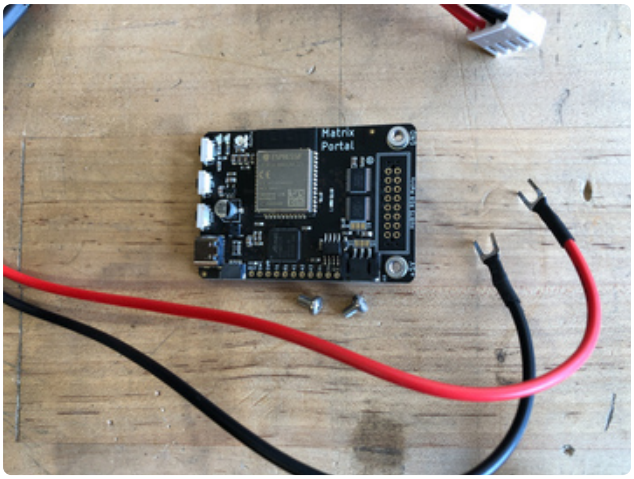


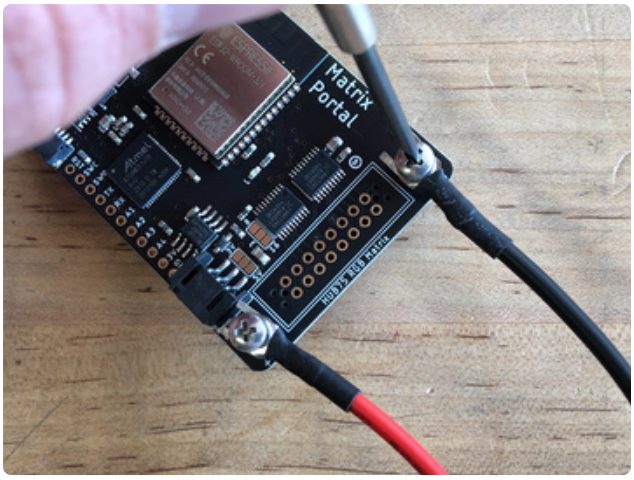


Power Prep

The MatrixPortal supplies power to the matrix display panel via two standoffs. These come with protective tape applied (part of our manufacturing process) which MUST BE REMOVED!

Use some tweezers or a fingernail to remove the two amber circles.

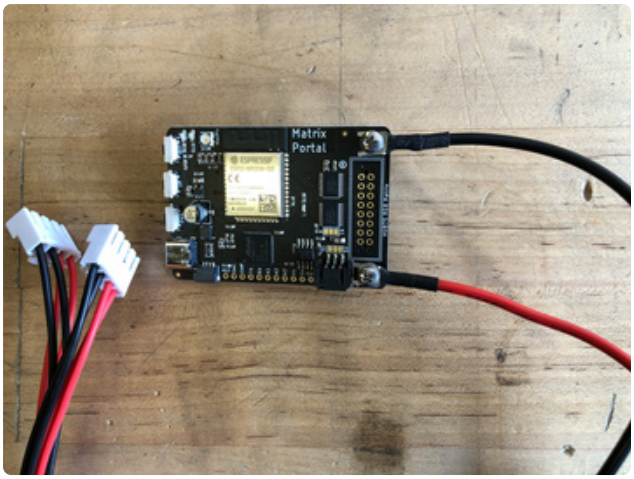


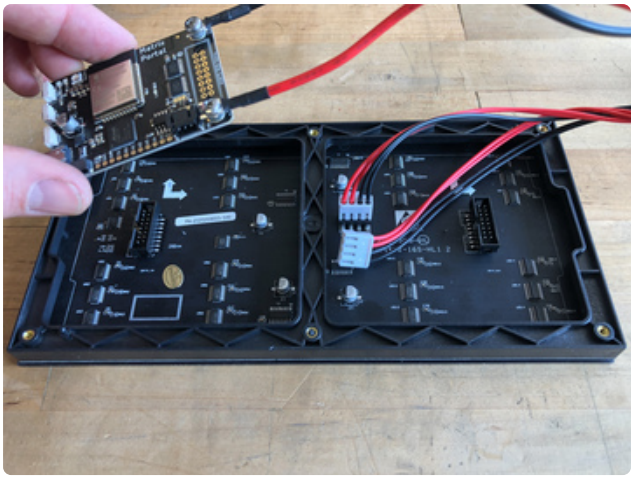


Power Terminals

Next, screw in the spade connectors to the corresponding standoff.

red wire goes to **+5V**
black wire goes to **GND**



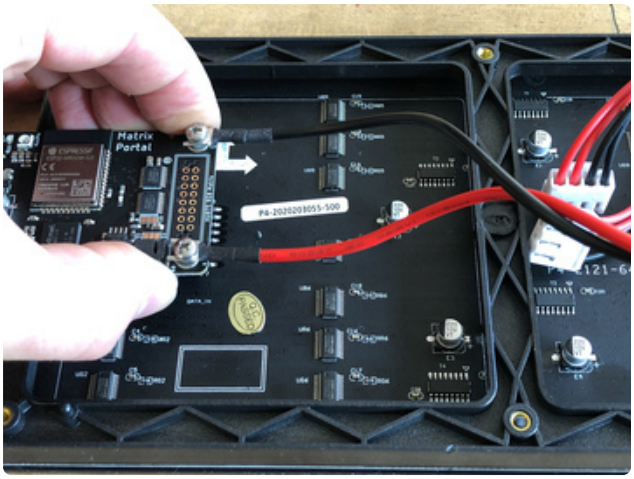


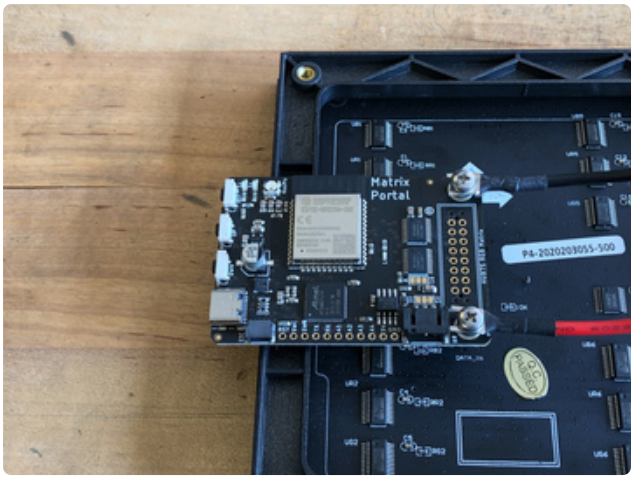
Panel Power

Plug either one of the four-conductor power plugs into the power connector pins on the panel. The plug can only go in one way, and that way is marked on the board's silkscreen.

Dual Matrix Setup

If you're planning to use a 64x64 matrix, follow these instructions on soldering the Address E Line jumper (https://adafru.it/OdJ).

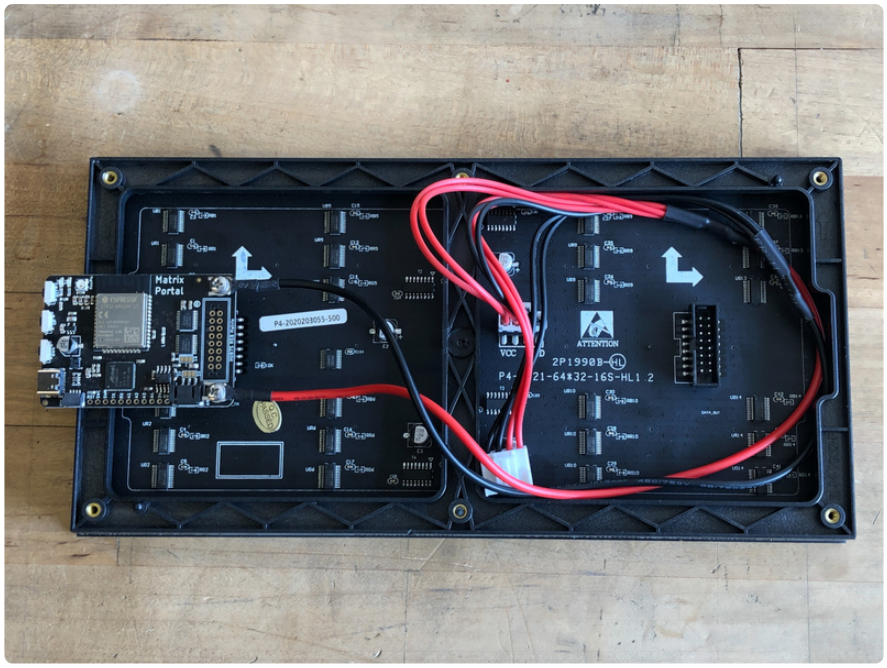




Board Connection

Now, plug the board into the left side shrouded 8x2 connector as shown. The orientation matters, so take a moment to confirm that the **white indicator arrow on the matrix panel is oriented pointing up and right** as seen here and the MatrixPortal overhangs the edge of the panel when connected. This allows you to use the edge buttons from the front side.

Check nothing is impeding the board from plugging in firmly. If there's a plastic nub on the matrix that's keeping the Portal from sitting flat, cut it off with diagonal cutters



For info on adding LED diffusion acrylic, see the page LED Matrix Diffuser.

Install CircuitPython

CircuitPython (https://adafru.it/tB7) is a derivative of MicroPython (https://adafru.it/BeZ) designed to simplify experimentation and education on low-cost microcontrollers. It makes it easier than ever to get prototyping by requiring no upfront desktop software downloads. Simply copy and edit files on the **CIRCUITPY** drive to iterate.

Set up CircuitPython Quick Start!

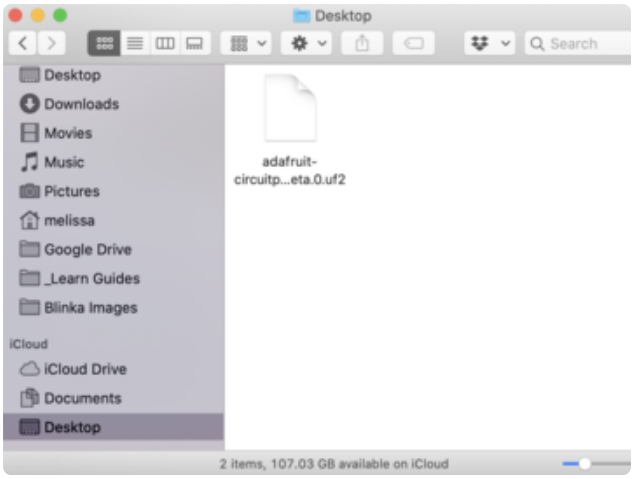
Follow this quick step-by-step for super-fast Python power :)

Download the latest version of CircuitPython for this board via circuitpython.org

https://adafru.it/Nte

Further Information

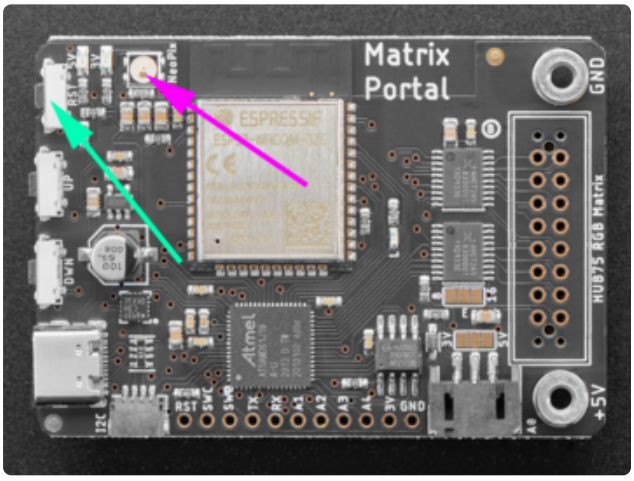
For more detailed info on installing CircuitPython, check out Installing CircuitPython (https://adafru.it/Amd).



Click the link above and download the latest UF2 file.

Download and save it to your desktop (or wherever is handy).

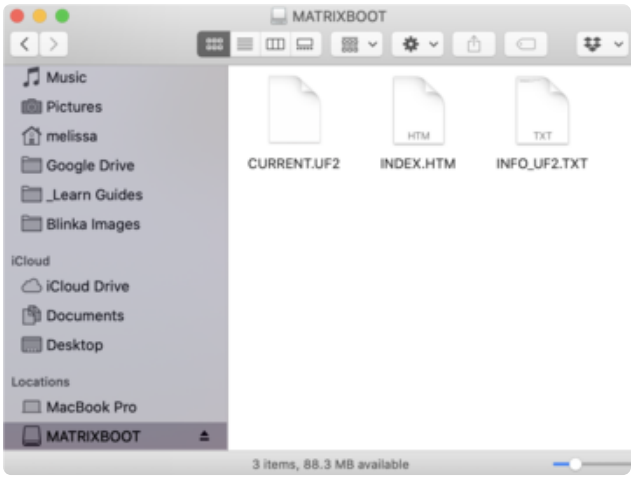
Plug your MatrixPortal M4 into your computer using a known-good USB cable.



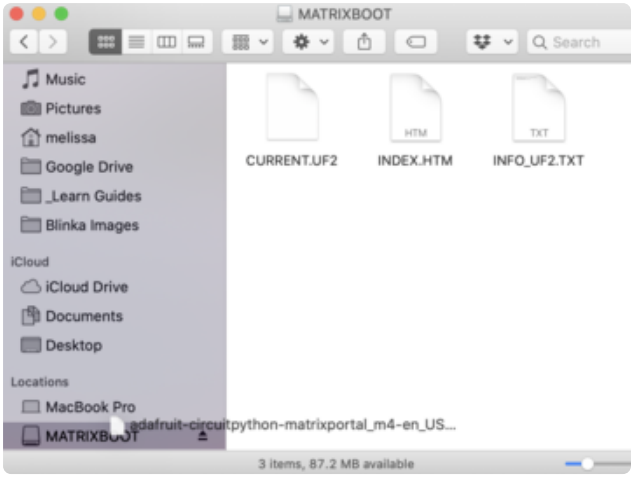
A lot of people end up using charge-only USB cables and it is very frustrating! So make sure you have a USB cable you know is good for data sync.

Double-click the **Reset** button (indicated by the green arrow) on your board, and you will see the NeoPixel RGB LED (indicated by the magenta arrow) turn green. If it turns red, check the USB cable, try another USB port, etc.

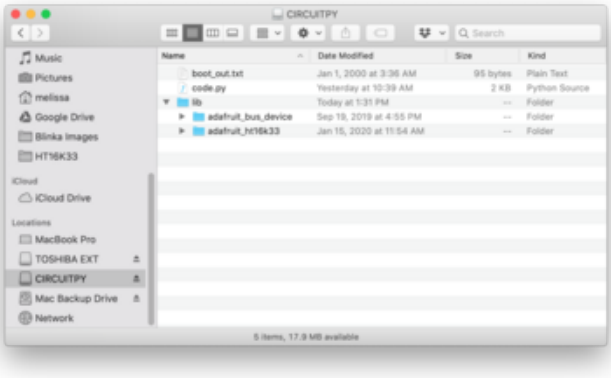
If double-clicking doesn't work the first time, try again. Sometimes it can take a few tries to get the rhythm right!



You will see a new disk drive appear called **MATRIXBOOT**.



Drag the **adafruit_circuitpython_etc.uf2** file to **MATRIXBOOT.**



The LED will flash. Then, the **MATRIXBOOT** drive will disappear and a new disk drive called **CIRCUITPY** will appear.

That's it, you're done! :)

Install Code and Graphics

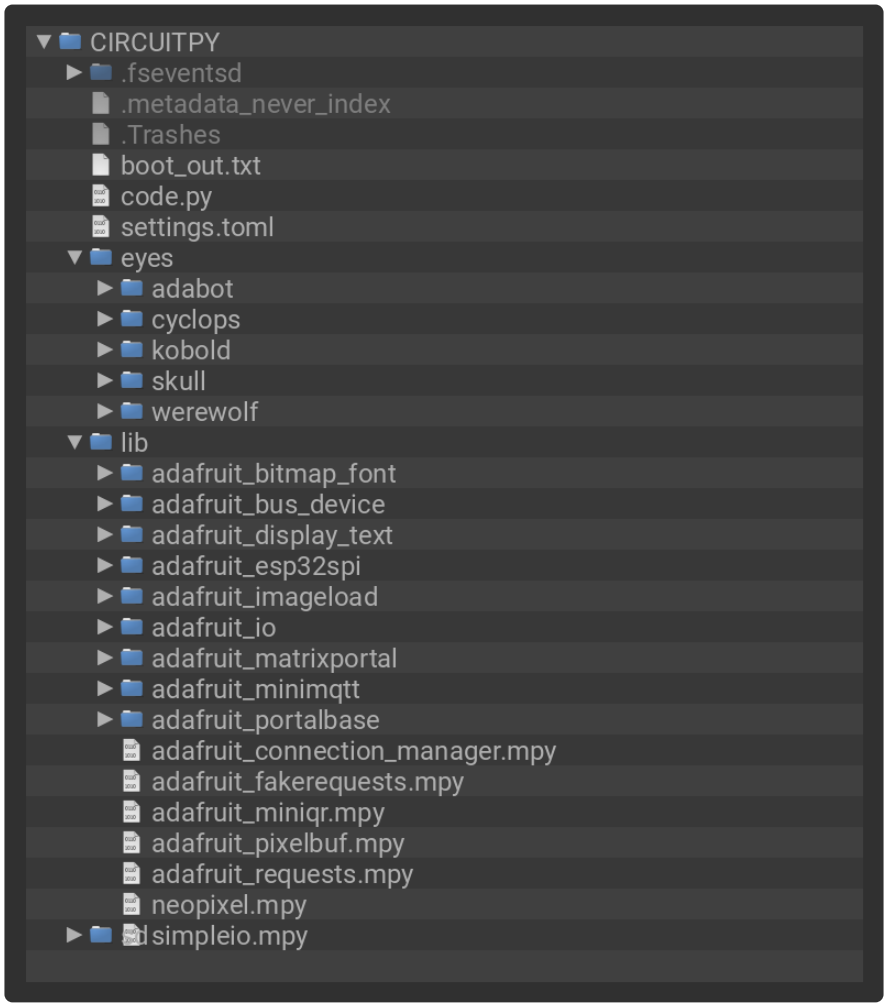
Back up any existing code or files you want to keep from your Matrix Portal CIRCUITPY drive.

Installing Project Code

To use with CircuitPython, you need to first install a few libraries, into the lib folder on your **CIRCUITPY** drive. Then you need to update **code.py** with the example script.

Thankfully, we can do this in one go. In the example below, click the **Download Project Bundle** button below to download the necessary libraries and the **code.py** file in a zip file. Extract the contents of the zip file, open the directory **Matrix_Portal_Eyes/** and then click on the directory that matches the version of CircuitPython you're using and copy the contents of that directory to your **CIRCUITPY** drive.

Your **CIRCUITPY** drive should now look similar to the following image:




```
# SPDX-FileCopyrightText: 2020 Phillip Burgess for Adafruit Industries
#
# SPDX-License-Identifier: MIT

"""
RASTER EYES for Adafruit Matrix Portal: animated spooky eyes.
"""

# pylint: disable=import-error
```

```
import math
import random
import time
import displayio
import adafruit_imageload
from adafruit_matrixportal.matrix import Matrix

# TO LOAD DIFFERENT EYE DESIGNS: change the middle word here (between
# 'eyes.' and '.data') to one of the folder names inside the 'eyes' folder:
from eyes.werewolf.data import EYE_DATA
#from eyes.cyclops.data import EYE_DATA
#from eyes.kobold.data import EYE_DATA
#from eyes.adabot.data import EYE_DATA
#from eyes.skull.data import EYE_DATA

# UTILITY FUNCTIONS AND CLASSES --------------------------------------------

# pylint: disable=too-few-public-methods
class Sprite(displayio.TileGrid):
    """Single-tile-with-bitmap TileGrid subclass, adds a height element
       because TileGrid doesn't appear to have a way to poll that later,
       object still functions in a displayio.Group.
    """
    def __init__(self, filename, transparent=None):
        """Create Sprite object from color-paletted BMP file, optionally
           set one color to transparent (pass as RGB tuple or list to locate
           nearest color, or integer to use a known specific color index).
        """
        bitmap, palette = adafruit_imageload.load(
            filename, bitmap=displayio.Bitmap, palette=displayio.Palette)
        if isinstance(transparent, (tuple, list)): # Find closest RGB match
            closest_distance = 0x1000000           # Force first match
            for color_index, color in enumerate(palette): # Compare each...
                delta = (transparent[0] - ((color >> 16) & 0xFF),
                         transparent[1] - ((color >> 8) & 0xFF),
                         transparent[2] - (color & 0xFF))
                rgb_distance = (delta[0] * delta[0] +
                                delta[1] * delta[1] +
                                delta[2] * delta[2]) # Actually dist^2
                if rgb_distance < closest_distance:  # but adequate for
                    closest_distance = rgb_distance  # compare purposes,
                    closest_index = color_index      # no sqrt needed
            palette.make_transparent(closest_index)
        elif isinstance(transparent, int):
            palette.make_transparent(transparent)
        super(Sprite, self).__init__(bitmap, pixel_shader=palette)


# ONE-TIME INITIALIZATION --------------------------------------------------

MATRIX = Matrix(bit_depth=6)
DISPLAY = MATRIX.display

# Order in which sprites are added determines the 'stacking order' and
# visual priority. Lower lid is added before the upper lid so that if they
# overlap, the upper lid is 'on top' (e.g. if it has eyelashes or such).
SPRITES = displayio.Group()
SPRITES.append(Sprite(EYE_DATA['eye_image'])) # Base image is opaque
SPRITES.append(Sprite(EYE_DATA['lower_lid_image'], EYE_DATA['transparent']))
SPRITES.append(Sprite(EYE_DATA['upper_lid_image'], EYE_DATA['transparent']))
SPRITES.append(Sprite(EYE_DATA['stencil_image'], EYE_DATA['transparent']))
DISPLAY.root_group = SPRITES

EYE_CENTER = ((EYE_DATA['eye_move_min'][0] +           # Pixel coords of eye
               EYE_DATA['eye_move_max'][0]) / 2,       # image when centered
              (EYE_DATA['eye_move_min'][1] +           # ('neutral' position)
               EYE_DATA['eye_move_max'][1]) / 2)
EYE_RANGE = (abs(EYE_DATA['eye_move_max'][0] -         # Max eye image motion
                 EYE_DATA['eye_move_min'][0]) / 2,     # delta from center
```

```
             abs(EYE_DATA['eye_move_max'][1] -
                 EYE_DATA['eye_move_min'][1]) / 2)
UPPER_LID_MIN = (min(EYE_DATA['upper_lid_open'][0],    # Motion bounds of
                     EYE_DATA['upper_lid_closed'][0]), # upper and lower
                 min(EYE_DATA['upper_lid_open'][1],    # eyelids
                     EYE_DATA['upper_lid_closed'][1]))
UPPER_LID_MAX = (max(EYE_DATA['upper_lid_open'][0],
                     EYE_DATA['upper_lid_closed'][0]),
                 max(EYE_DATA['upper_lid_open'][1],
                     EYE_DATA['upper_lid_closed'][1]))
LOWER_LID_MIN = (min(EYE_DATA['lower_lid_open'][0],
                     EYE_DATA['lower_lid_closed'][0]),
                 min(EYE_DATA['lower_lid_open'][1],
                     EYE_DATA['lower_lid_closed'][1]))
LOWER_LID_MAX = (max(EYE_DATA['lower_lid_open'][0],
                     EYE_DATA['lower_lid_closed'][0]),
                 max(EYE_DATA['lower_lid_open'][1],
                     EYE_DATA['lower_lid_closed'][1]))
EYE_PREV = (0, 0)
EYE_NEXT = (0, 0)
MOVE_STATE = False                                    # Initially stationary
MOVE_EVENT_DURATION = random.uniform(0.1, 3)          # Time to first move
BLINK_STATE = 2                                       # Start eyes closed
BLINK_EVENT_DURATION = random.uniform(0.25, 0.5)      # Time for eyes to open
TIME_OF_LAST_MOVE_EVENT = TIME_OF_LAST_BLINK_EVENT = time.monotonic()


# MAIN LOOP -----------------------------------------------------------------

while True:
    NOW = time.monotonic()

    # Eye movement --------------------------------------------------------

    if NOW - TIME_OF_LAST_MOVE_EVENT > MOVE_EVENT_DURATION:
        TIME_OF_LAST_MOVE_EVENT = NOW # Start new move or pause
        MOVE_STATE = not MOVE_STATE   # Toggle between moving & stationary
        if MOVE_STATE:                # Starting a new move?
            MOVE_EVENT_DURATION = random.uniform(0.08, 0.17) # Move time
            ANGLE = random.uniform(0, math.pi * 2)
            EYE_NEXT = (math.cos(ANGLE) * EYE_RANGE[0], # (0,0) in center,
                        math.sin(ANGLE) * EYE_RANGE[1]) # NOT pixel coords
        else:                         # Starting a new pause
            MOVE_EVENT_DURATION = random.uniform(0.04, 3)    # Hold time
            EYE_PREV = EYE_NEXT

    # Fraction of move elapsed (0.0 to 1.0), then ease in/out 3*e^2-2*e^3
    RATIO = (NOW - TIME_OF_LAST_MOVE_EVENT) / MOVE_EVENT_DURATION
    RATIO = 3 * RATIO * RATIO - 2 * RATIO * RATIO * RATIO
    EYE_POS = (EYE_PREV[0] + RATIO * (EYE_NEXT[0] - EYE_PREV[0]),
               EYE_PREV[1] + RATIO * (EYE_NEXT[1] - EYE_PREV[1]))

    # Blinking ------------------------------------------------------------

    if NOW - TIME_OF_LAST_BLINK_EVENT > BLINK_EVENT_DURATION:
        TIME_OF_LAST_BLINK_EVENT = NOW # Start change in blink
        BLINK_STATE += 1               # Cycle paused/closing/opening
        if BLINK_STATE == 1:           # Starting a new blink (closing)
            BLINK_EVENT_DURATION = random.uniform(0.03, 0.07)
        elif BLINK_STATE == 2:         # Starting de-blink (opening)
            BLINK_EVENT_DURATION *= 2
        else:                          # Blink ended,
            BLINK_STATE = 0            # paused
            BLINK_EVENT_DURATION = random.uniform(BLINK_EVENT_DURATION * 3, 4)

    if BLINK_STATE: # Currently in a blink?
        # Fraction of closing or opening elapsed (0.0 to 1.0)
        RATIO = (NOW - TIME_OF_LAST_BLINK_EVENT) / BLINK_EVENT_DURATION
        if BLINK_STATE == 2:    # Opening
```

```
            RATIO = 1.0 - RATIO # Flip ratio so eye opens instead of closes
    else:           # Not blinking
        RATIO = 0

    # Eyelid tracking -------------------------------------------------------

    # Initial estimate of 'tracked' eyelid positions
    UPPER_LID_POS = (EYE_DATA['upper_lid_center'][0] + EYE_POS[0],
                     EYE_DATA['upper_lid_center'][1] + EYE_POS[1])
    LOWER_LID_POS = (EYE_DATA['lower_lid_center'][0] + EYE_POS[0],
                     EYE_DATA['lower_lid_center'][1] + EYE_POS[1])
    # Then constrain these to the upper/lower lid motion bounds
    UPPER_LID_POS = (min(max(UPPER_LID_POS[0],
                             UPPER_LID_MIN[0]), UPPER_LID_MAX[0]),
                     min(max(UPPER_LID_POS[1],
                             UPPER_LID_MIN[1]), UPPER_LID_MAX[1]))
    LOWER_LID_POS = (min(max(LOWER_LID_POS[0],
                             LOWER_LID_MIN[0]), LOWER_LID_MAX[0]),
                     min(max(LOWER_LID_POS[1],
                             LOWER_LID_MIN[1]), LOWER_LID_MAX[1]))
    # Then interpolate between bounded tracked position to closed position
    UPPER_LID_POS = (UPPER_LID_POS[0] + RATIO *
                     (EYE_DATA['upper_lid_closed'][0] - UPPER_LID_POS[0]),
                     UPPER_LID_POS[1] + RATIO *
                     (EYE_DATA['upper_lid_closed'][1] - UPPER_LID_POS[1]))
    LOWER_LID_POS = (LOWER_LID_POS[0] + RATIO *
                     (EYE_DATA['lower_lid_closed'][0] - LOWER_LID_POS[0]),
                     LOWER_LID_POS[1] + RATIO *
                     (EYE_DATA['lower_lid_closed'][1] - LOWER_LID_POS[1]))

    # Move eye sprites ------------------------------------------------------

    SPRITES[0].x, SPRITES[0].y = (int(EYE_CENTER[0] + EYE_POS[0] + 0.5),
                                  int(EYE_CENTER[1] + EYE_POS[1] + 0.5))
    SPRITES[2].x, SPRITES[2].y = (int(UPPER_LID_POS[0] + 0.5),
                                  int(UPPER_LID_POS[1] + 0.5))
    SPRITES[1].x, SPRITES[1].y = (int(LOWER_LID_POS[0] + 0.5),
                                  int(LOWER_LID_POS[1] + 0.5))
```

And that's it! If all the right files are copied over, the creature eyes should **begin running automatically**.

If you want to **select a different creature** (a few designs are included) or **create your own**, that's explained next...

Ready-Made Creatures

To **select among different ready-made creatures**, edit the file **code.py** on the **CIRCUITPY** drive, using your text editor of preference.

A few lines into the code you'll see several references to **EYE_DATA**. Most of these lines are "commented out" — they have no effect on the CircuitPython code. A "`#`" character indicates the start of a comment. But **one** of these lines is enabled.

```
from eyes.werewolf.data import EYE_DATA
#from eyes.cyclops.data import EYE_DATA
#from eyes.kobold.data import EYE_DATA
```

All you need to do is comment out the currently-active line (the “werewolf” line above) by **adding** a `#`.

Then **remove** the `#` from the line you want to enable. Only **one** of these lines should be active at a time.

Save the changes to the file, and the code should **restart automatically**, provided you did the comment/uncomment change correctly.

Our default **werewolf** design is active with the `eyes.werewolf.data` line enabled.

Werewolves have been a thing this year, with the Halloween full Moon (https://adafru.it/NB7) and all.

Enable the `eyes.cyclops.data` line to bring up this icky single eye... demonstrating that it doesn’t always need to be two (in theory it could do more, if you can design something legible at that resolution).

And this creature comes from the `eyes.kobold.data` line.

In traditional Germanic folklore, kobolds are sort of gnome-like creatures. Dungeons & Dragons popularized the idea of kobolds as small lizard people. I’m okay with that, lizards are cool.

Making New Creatures

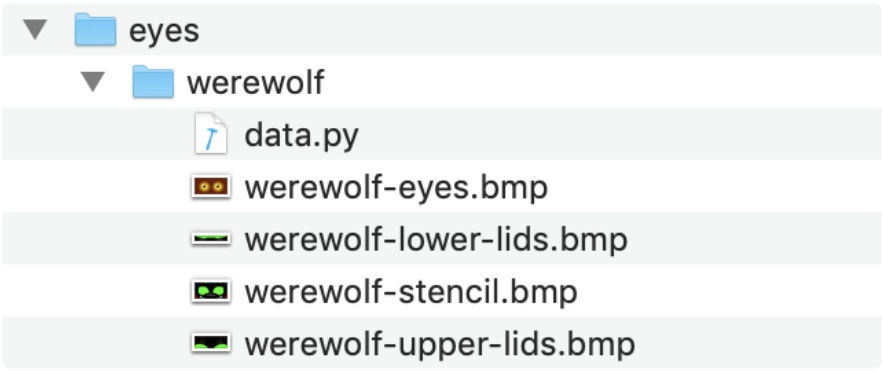
Let's suppose you want to go all-out and hatch some creatures of your own.

You will need:

- Image editing software that can create BMP files (I use Photoshop, but even most basic paint programs will do...and if they don't save BMP directly, you can find free image conversion utilities to help)
- An understanding of image coordinate systems (explained below)
- A text editor (we like Mu (https://adafru.it/LEQ) for editing CircuitPython projects, but most anything will do)

With the eyes project already installed and running...on the CIRCUITPY drive, each sub-folder inside the "**eyes**" folder contains 5 files:

- Four BMP image files
- data.py, a bit of Python code describing how the images work together



If making a **new creature**, it's easiest to start by **duplicating** one of the **existing folders** and giving it a descriptive name, e.g. "gillman". Then the files within can be edited and renamed as needed.

To **activate** your new creature, remember to add a line in **code.py**, telling it to import your new design...as was done with the ready-made creatures. This is based on the **name of the folder**, e.g.:

```
from eyes.gillman.data import EYE_DATA
```

(And then **comment out** any/all inactive creatures in **code.py**. Only **one** should be active.)

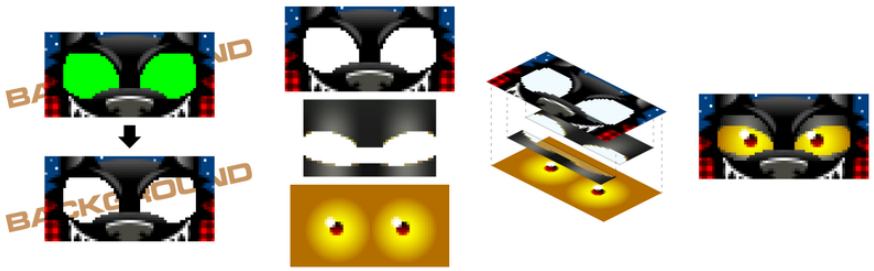
Graphics and Coordinates

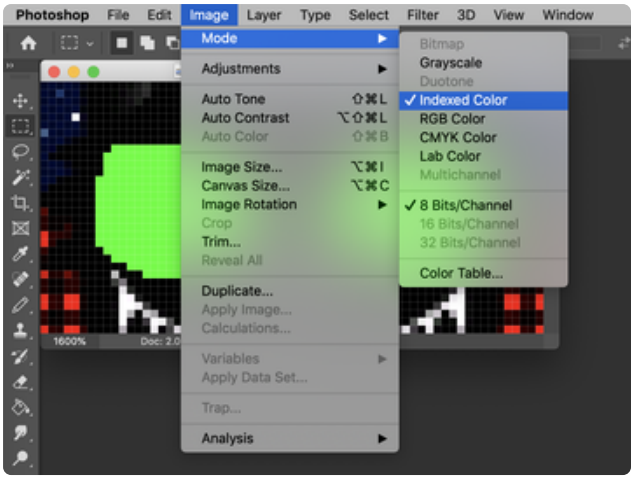
Some of the images are partly **transparent**. The animation code “stacks” these one atop the next, and shuffles them around to create the looking-and-blinking effect.

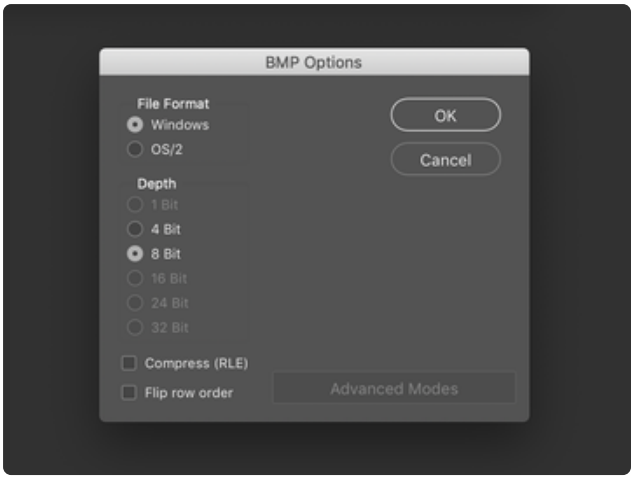
The **top-most image** in this stack depicts most of the monster’s **face**, with **cut-outs** for the eyes (or single eye, in the case of the cyclops example). For the **werewolf** example, this image is called **werewolf-stencil.bmp**. This image is always **the same size as the LED matrix** (64x32 pixels in the examples) and **never moves**. The eye holes are bright green in this image (this color was chosen because it’s not used anywhere else in the werewolf set), but in other examples are bright red (for similar reasons).

Below this are **two images** for the **top and bottom eyelids**...same deal, bright green represents pixels that will be **transparent** in the final “stack.”

The **bottom-most** image is the creature’s **eyes**...the whites, pupils and so forth. This image does not use the transparent color, since there’s nothing below to show through. It needs to be **bigger than just the eye holes**, since it will be moving around and we want something to always be filling those pixels.







The images all need to be **8-bit** "indexed color" or "paletted" **BMP** files. 24-bit BMPs, and other image formats, **will not work** with this code.

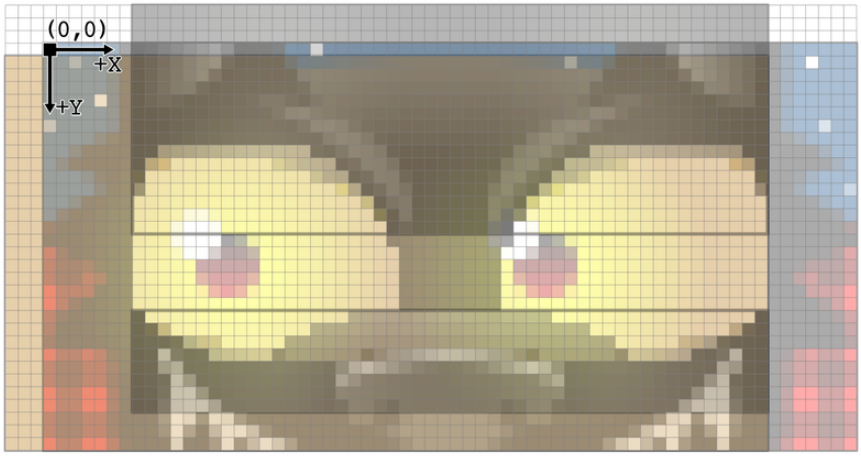
Here's an image being converted and saved from Photoshop, but the same functionality is available in many image editors, or there are free tools like ImageMagick (https://adafru.it/NBh).

Your images might appear washed-out and too bright on the matrix...the code that handles the LED matrix doesn't yet support gamma correction. You can compensate for this somewhat in your image editor...try adjusting the gamma (the middle value in Photoshop's "Levels" command) to 0.4 or so. **Keep a copy of your original art around!** Do this level adjustment only as a last step before saving. It's a destructive process and you might be testing several iterations to get the images just right.

Next is to figure out the pixel coordinates where things will be moving.

Remember that the frontmost "stencil" image is always the same size as the matrix, so that's a good point of reference. The **top left pixel** of this image has coordinates **(0, 0)**. Moving **right**, **X** (the first value in the coordinate pair) **increases by 1** for each pixel, or **-1** to the **left**. Moving **down**, **Y** (second value) increases by 1, or -1 for up.

Sometimes the images will be moved to negative coordinates. This is normal and okay! The graphics are automatically "clipped" to the matrix boundaries as needed.



So now, edit the file **data.py** using your text editor of preference. Here's what that file looks like for the werewolf:

Coding Positions and Movement

```
""" Configuration data for the werewolf eyes """
EYE_PATH = __file__[:__file__.rfind('/') + 1]
EYE_DATA = {
    'eye_image'         : EYE_PATH + 'werewolf-eyes.bmp',
    'upper_lid_image'   : EYE_PATH + 'werewolf-upper-lids.bmp',
    'lower_lid_image'   : EYE_PATH + 'werewolf-lower-lids.bmp',
    'stencil_image'     : EYE_PATH + 'werewolf-stencil.bmp',
    'transparent'       : (0, 255, 0),
    'eye_move_min'      : (-3, -5),
    'eye_move_max'      : (7, 6),
    'upper_lid_open'    : (7, -4),
    'upper_lid_center'  : (7, -1),
    'upper_lid_closed'  : (7, 8),
    'lower_lid_open'    : (7, 22),
    'lower_lid_center'  : (7, 21),
    'lower_lid_closed'  : (7, 17),
}
```

Keep in mind, **this is Python code**, and so it's going to be **strict** about syntax! Make sure strings are quoted, there's a comma at the end of each line in the table and so forth. If your new creature fails to run, the problem may be with the file syntax...keep a serial connection open to the board and see what it says.

The **EYE_PATH** line is just weird Python syntax for "wherever this file is located, we want to locate things in the same directory." It avoids us having to specify an absolute path to every image file.

EYE_DATA is a Python dictionary — it contains pairs of keys and values, each separated by a colon. The **key strings must remain unchanged**. These are the names that the main Python code (in code.py) will be looking for. **Edit only the values.**

The first four items in this dictionary specify the **BMP image files** used for the different layers of the animation, as described earlier. EYE_PATH is mentioned here to indicate "these images are in the same directory as this file."

The next item, `'transparent'`, is the RGB color value (red, green, blue) that will "show through" in these images as they're stacked. It's green for the werewolf, red for other examples.

`'eye_move_min'` is the leftmost and topmost position of the eye image when looking directly left and up, respectively. `'eye_move_max'` is the rightmost and bottom-most positions. These two points describe a rectangle...but the actual eye movement will be constrained to an ellipse inside this rectangle, so don't worry if your pupils go out of bounds when at these two points...it's really the "compass points" left, top, right and bottom that we care about.

The next three items, `'upper_lid_open',` `'upper_lid_center'` and `'upper_lid_closed'` are the **top-left pixel coordinates** of the **upper** eyelid image **relative to the matrix** (or the topmost stencil image), at the eyelid's most-open position, in a neutral (eyes centered) position, and at its lowest position (when blinking). Having three values here (rather than just two) allows the eyelids to "track" the movement of the eyes, which is a neat thing that eyelids really do. The `'upper_lid_closed'` value usually won't completely cover the eye, unless you're specifically aiming for that look. It's normal for the upper and lower eyelids to meet part way. The X values for all these coordinates will usually be the same.

Last three items, `'lower_lid_open'` , `'lower_lid_center'` and `'lower_lid_closed'` are similar, but for the bottom eyelid. Whereas the upper eyelid will usually have increasing Y values for each (going top to bottom), the lower eyelid will usually have decreasing Y values.

Even with careful planning, the various eye images might not be the right sizes or end up in the intended positions on the first try. It's perfectly normal to take a few iterations through this file to get all the graphics lined up and moving as intended.

If you're editing data.py but not seeing any changes: make sure you're also loading the correct file in **code.py** (as shown earlier on this page).