PyBadge Conference Badge With Unicode Fonts

Created by Melissa LeBlanc-Williams

https://learn.adafruit.com/pybadge-conference-badge-multi-language-unicode-fonts

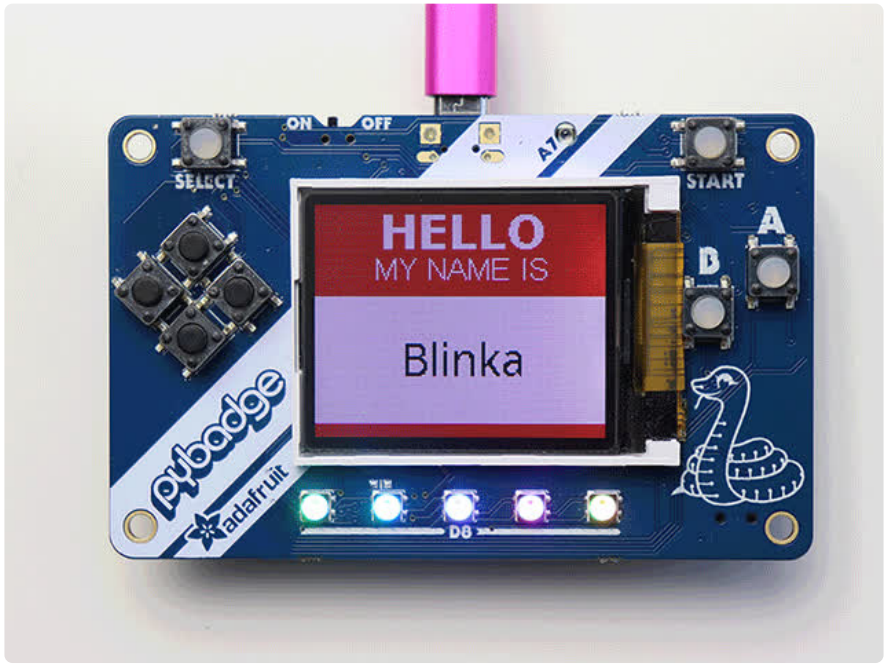
Last updated on 2024-06-03 02:46:27 PM EDT

Table of Contents

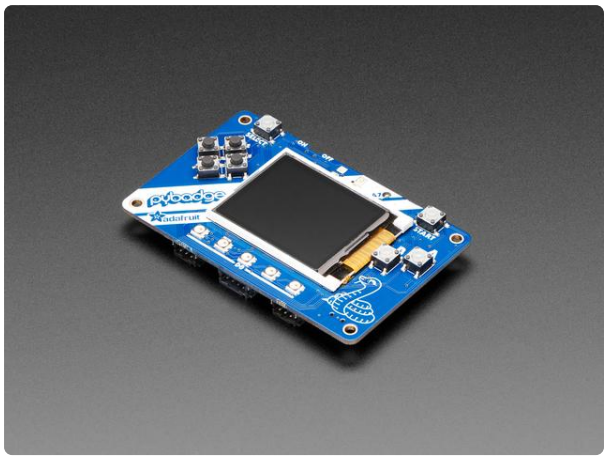
Overview

Get ready for your next conference with the PyBadge Conference Badge. With this simple project, not only can you show off some of your basic details for people to contact you, but you can also control the NeoPixels at the bottom to draw people's attention to your badge.

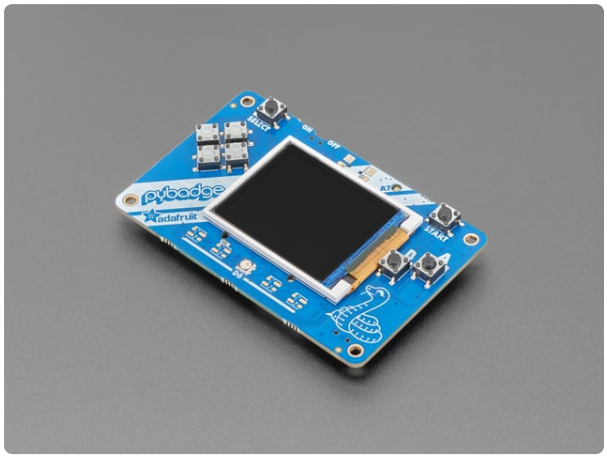
This project displays a simple name tag that is drawn using code, so it is easy to customize without the need to do any graphical editing. Additionally, since CircuitPython is capable of displaying unicode, fonts are very customizable.



Parts



Adafruit PyBadge for MakeCode Arcade, CircuitPython, or Arduino
What's the size of a credit card and can run CircuitPython, MakeCode Arcade or Arduino? That's right, its the Adafruit PyBadge! We wanted to see how much we...
https://www.adafruit.com/product/4200



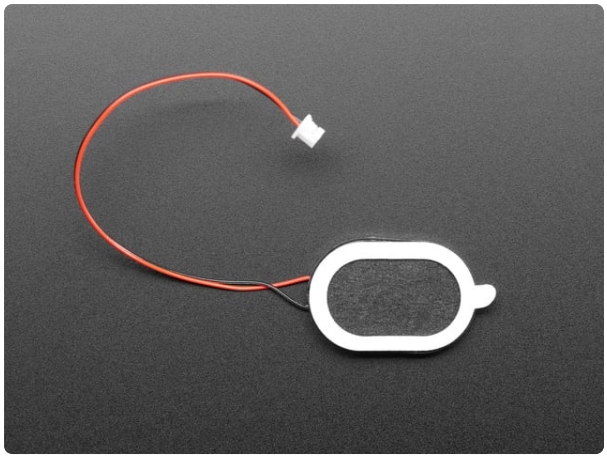
Adafruit PyBadge LC - MakeCode Arcade, CircuitPython, or Arduino
What's the size of a credit card and can run CircuitPython, MakeCode Arcade or Arduino even when you're on a budget? That's right, it's the Adafruit...
https://www.adafruit.com/product/3939

Accessories

Be sure to grab a lanyard so you can easily show off your badge.

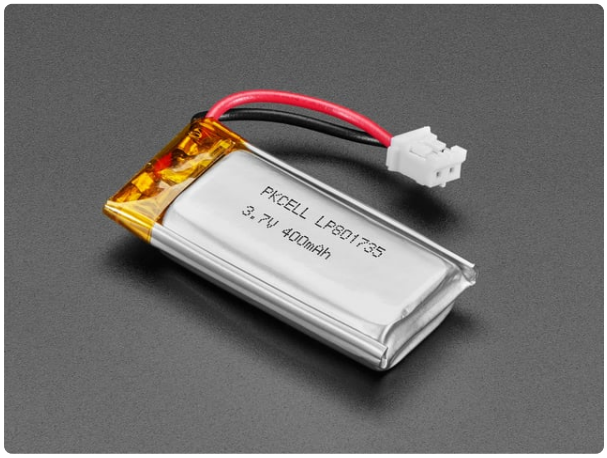
Adafruit Circuit Playground Lanyard
We've got our Circuit Playground friends on lunchboxes,
https://www.adafruit.com/product/3987

This project doesn't use the speaker, but they're inexpensive and very easy to add in case you decide to get creative and add sound.



Mini Oval Speaker - 8 Ohm 1 Watt
Hear the good news! This wee speaker is a great addition to any audio project where you need 8 ohm impedance and 1W or less of power. We particularly like...
https://www.adafruit.com/product/3923

Also, be sure to pick up a couple of batteries in case they get low on power so you can just swap them out. The PyBadge comes with built-in charging circuitry, so they're very easy to charge.



Lithium Ion Polymer Battery Ideal For Feathers - 3.7V 400mAh
Lithium-ion polymer (also known as 'lipo' or 'lipoly') batteries are thin, light, and powerful. The output ranges from 4.2V when completely charged to 3.7V. This...
https://www.adafruit.com/product/3898

Setup

To get set up, we will need CircuitPython, a few libraries and the Conference Badge Source Code and fonts downloaded from Github.

Required CircuitPython Libraries

First, make sure you are running the latest version of Adafruit CircuitPython (https://adafru.it/Amd) for your board.

Next, you'll need to install the necessary libraries to use the hardware--carefully follow the steps to find and install these libraries from Adafruit's CircuitPython library bundle (https://adafru.it/zdx). Our introduction guide has a great page on how to install the library bundle (https://adafru.it/ABU) for both express and non-express boards.

Since the fonts take up a good deal of space, Adafruit recommends that you manually install only the necessary libraries from the bundle:

- **neopixel**
- **adafruit_bitmap_font**
- **adafruit_display_shapes**
- **adafruit_display_text**

Before continuing make sure your board's lib folder or root filesystem have the **neopixel**, **adafruit_bitmap_font**, **adafruit_display_shapes**, and **adafruit_display_text** files and folders copied over.

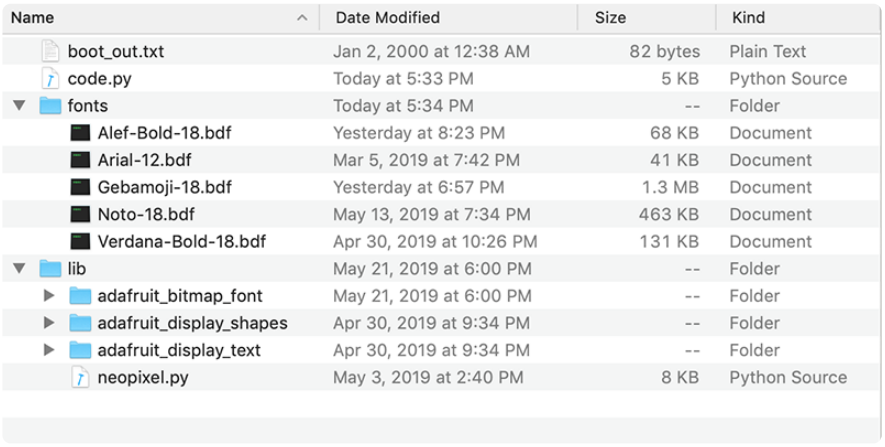
Conference Badge Source Code

Next, we'll continue by downloading the Source Code from Github.

Conference Badge Source Code

https://adafru.it/EUG

The fonts go in a subdirectory of your PyBadge's main **CIRCUITPY** flash drive named / **fonts**.



Here is the CircuitPython code in its entirety. If you click on Download Project Zip, it will download the fonts used in this tutorial also.

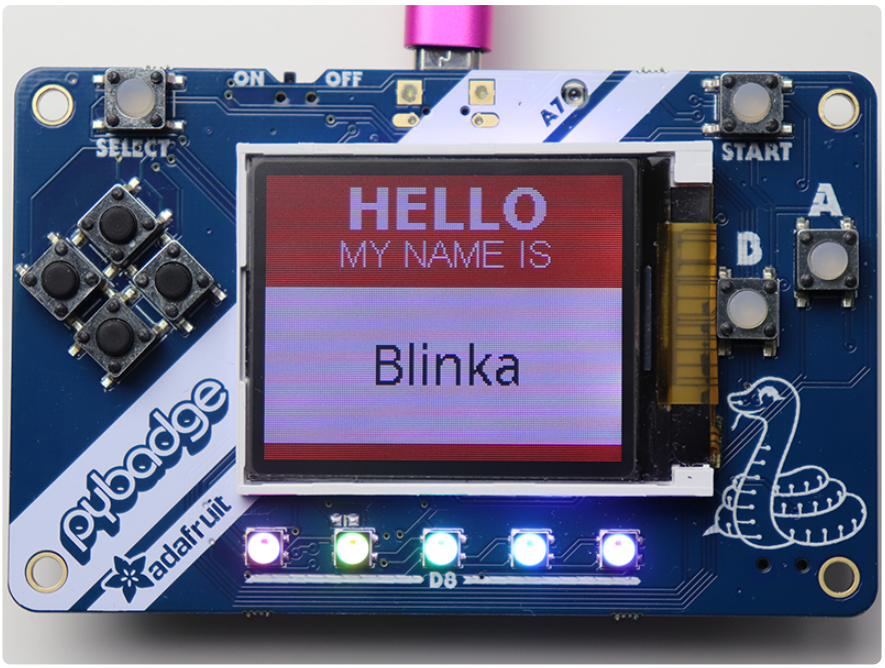
```
# SPDX-FileCopyrightText: 2019 Melissa LeBlanc-Williams for Adafruit Industries
#
# SPDX-License-Identifier: MIT

"""
This is a Conference Badge type Name Tag that is intended to be displayed on
the PyBadge. Feel free to customize it to your heart's content.
"""
```

```
from math import sqrt, cos, sin, radians
import board
from micropython import const
import displayio
import neopixel
from keypad import ShiftRegisterKeys, Event
from adafruit_display_shapes.rect import Rect
from adafruit_display_text.label import Label
from adafruit_bitmap_font import bitmap_font

# Button Constants
BUTTON_LEFT = const(7)
BUTTON_UP = const(6)
BUTTON_DOWN = const(5)
BUTTON_RIGHT = const(4)
BUTTON_SEL = const(3)
BUTTON_START = const(2)
BUTTON_A = const(1)
BUTTON_B = const(0)

# Customizations
HELLO_STRING = "HELLO"
MY_NAME_STRING = "MY NAME IS"
NAME_STRING = "Blinka"
NAME_FONTNAME = "/fonts/Noto-18.bdf"
NEOPIXEL_COUNT = 5
BACKGROUND_COLOR = 0xFF0000
FOREGROUND_COLOR = 0xFFFFFF
BACKGROUND_TEXT_COLOR = 0xFFFFFF
FOREGROUND_TEXT_COLOR = 0x000000

settings = {"brightness": 0.2, "direction": 1, "speed": 1}

# Define the NeoPixel
neopixels = neopixel.NeoPixel(
    board.NEOPIXEL,
    NEOPIXEL_COUNT,
    brightness=settings["brightness"],
    auto_write=False,
    pixel_order=neopixel.GRB,
)

# Define Events and Shift Register
latest_event = Event()
last_event = Event()

pad = ShiftRegisterKeys(
    clock=board.BUTTON_CLOCK,
    data=board.BUTTON_OUT,
    latch=board.BUTTON_LATCH,
    key_count=8,
    value_when_pressed=True,
    interval=0.1,
    max_events=1,
)

# Make the Display Background
splash = displayio.Group()
board.DISPLAY.root_group = splash

color_bitmap = displayio.Bitmap(160, 128, 1)
color_palette = displayio.Palette(1)
color_palette[0] = BACKGROUND_COLOR

bg_sprite = displayio.TileGrid(color_bitmap, pixel_shader=color_palette, x=0, y=0)
splash.append(bg_sprite)

# Draw a Foreground Rectangle where the name goes
rect = Rect(0, 50, 160, 70, fill=FOREGROUND_COLOR)
```

```
splash.append(rect)

# Load the Hello font
large_font_name = "/fonts/Verdana-Bold-18.bdf"
large_font = bitmap_font.load_font(large_font_name)
large_font.load_glyphs(HELLO_STRING.encode("utf-8"))

# Load the "My Name Is" font
small_font_name = "/fonts/Arial-12.bdf"
small_font = bitmap_font.load_font(small_font_name)
small_font.load_glyphs(MY_NAME_STRING.encode("utf-8"))

# Load the Name font
name_font_name = NAME_FONTNAME
name_font = bitmap_font.load_font(name_font_name)
name_font.load_glyphs(NAME_STRING.encode("utf-8"))

# Setup and Center the Hello Label
splash.append(
    Label(
        large_font,
        anchor_point=(0.5, 0.5),
        anchored_position=(board.DISPLAY.width // 2, 15),
        text=HELLO_STRING,
        color=BACKGROUND_TEXT_COLOR,
    )
)

# Setup and Center the "My Name Is" Label
splash.append(
    Label(
        small_font,
        anchor_point=(0.5, 0.5),
        anchored_position=(board.DISPLAY.width // 2, 35),
        text=MY_NAME_STRING,
        color=BACKGROUND_TEXT_COLOR,
    )
)

# Setup and Center the Name Label
splash.append(
    Label(
        name_font,
        anchor_point=(0.5, 0.5),
        anchored_position=(board.DISPLAY.width // 2, 85),
        text=NAME_STRING,
        color=FOREGROUND_TEXT_COLOR,
    )
)

# Remap the calculated rotation to 0 - 255
def remap(vector):
    return int(((255 * vector + 85) * 0.75) + 0.5)


# Calculate the Hue rotation starting with Red as 0 degrees
def rotate(degrees):
    cosA = cos(radians(degrees))
    sinA = sin(radians(degrees))
    red = cosA + (1.0 - cosA) / 3.0
    green = 1.0 / 3.0 * (1.0 - cosA) + sqrt(1.0 / 3.0) * sinA
    blue = 1.0 / 3.0 * (1.0 - cosA) - sqrt(1.0 / 3.0) * sinA
    return (remap(red), remap(green), remap(blue))


palette = []
pixels = []

# Generate a rainbow palette
```

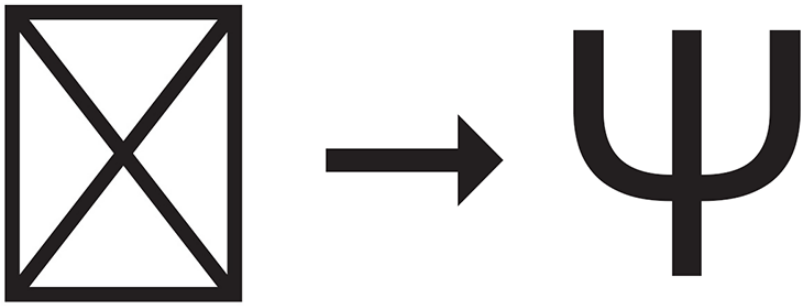
```
for degree in range(0, 360):
    color = rotate(degree)
    palette.append(color[0] << 16 | color[1] << 8 | color[2])

# Create the Pattern
for x in range(0, NEOPIXEL_COUNT):
    pixels.append(x * 360 // NEOPIXEL_COUNT)


def check_buttons(event):
    if event.key_number == BUTTON_RIGHT:
        settings["direction"] = -1
    elif event.key_number == BUTTON_LEFT:
        settings["direction"] = 1
    elif (event.key_number == BUTTON_UP) and settings["speed"] < 10:
        settings["speed"] += 1
    elif (event.key_number == BUTTON_DOWN) and settings["speed"] > 1:
        settings["speed"] -= 1
    elif (event.key_number == BUTTON_A) and settings["brightness"] < 0.5:
        settings["brightness"] += 0.025
    elif (event.key_number == BUTTON_B) and settings["brightness"] > 0.025:
        settings["brightness"] -= 0.025


# Main Loop
last_read = 0
while True:
    for color in range(0, 360, settings["speed"]):
        for index in range(0, NEOPIXEL_COUNT):
            palette_index = pixels[index] + color * settings["direction"]
            if palette_index >= 360:
                palette_index -= 360
            elif palette_index < 0:
                palette_index += 360
            neopixels[index] = palette[palette_index]
        neopixels.show()
        neopixels.brightness = settings["brightness"]
        pad.events.get_into(latest_event)
        if latest_event.pressed and latest_event.key_number !=
last_event.key_number:
            check_buttons(latest_event)
            last_event = latest_event
            latest_event = Event(
                key_number=8
            )  # An imaginary key number that doesn't exist!
```

After copying everything over, your PyBadge should display a Nametag with Blinka on it and the neopixels should animate. In the next few sections, we'll go over the Fonts and how to make changes.



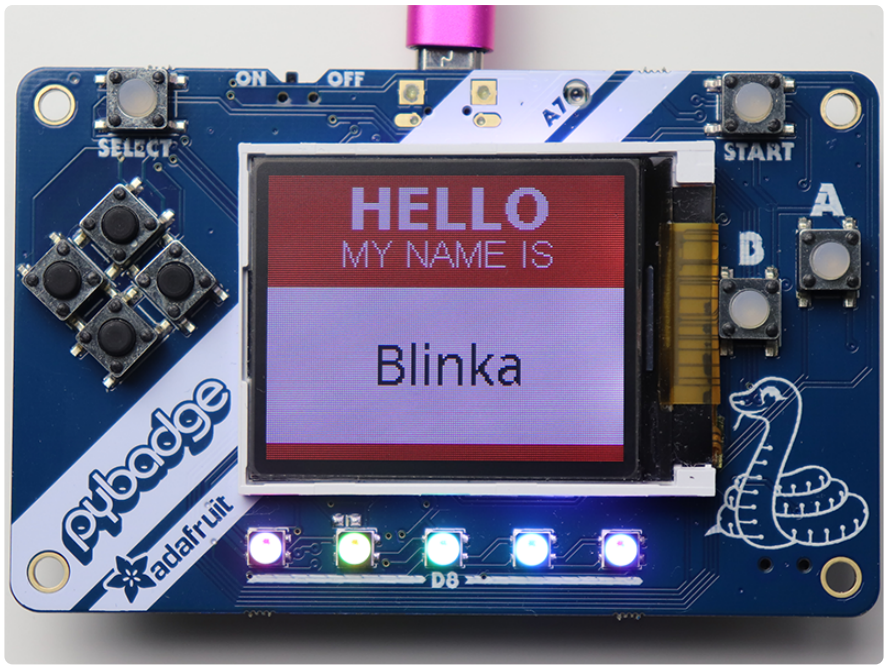
Multilingual Fonts

In order to provide as much language support in a single font, we looked at quite a few different ones. We decided to choose one of the Google Noto fonts. The specific font is the Noto Sans Display font due to its ability to support over 500 languages through its support of the Latin, Greek, and Cyrillic character sets. It is called a Noto Font because the goal of the font was to avoid needing to display the placeholder blocks that represent the missing characters. These placeholder blocks are referred to as "Tofu", so the name of the font is short for No Tofu.

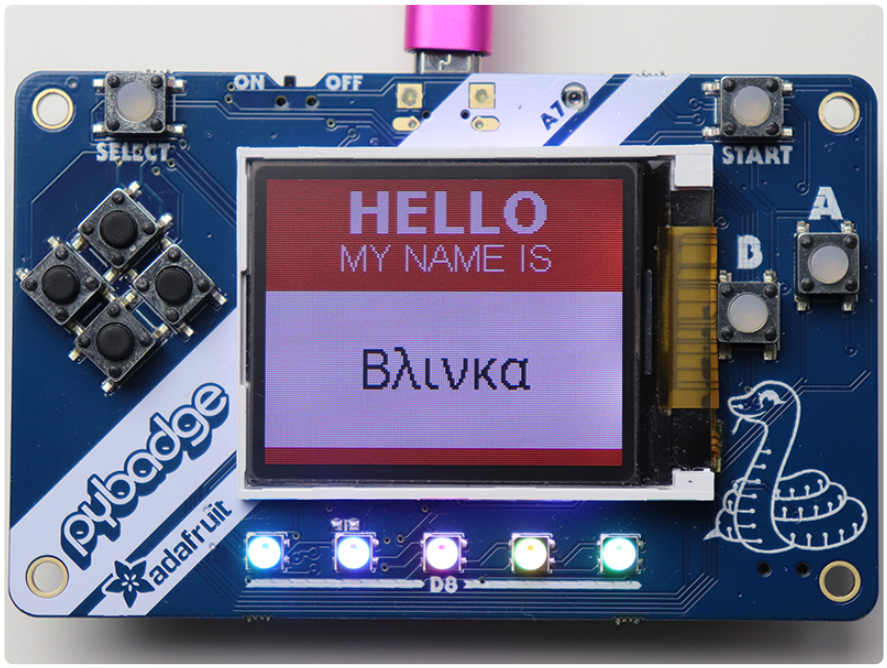


If you would like to make your own custom font, be sure to check out our Custom Fonts for CircuitPython Displays guide (https://adafru.it/E7E). For the fonts in this guide, we used otf2bdf.

For the basic Badge, we have a simple Hello, My Name Is Blinka. You can change the name "Blinka" to whatever your name is.



But what if your name has non-English characters? If your name has accents, that's easy since the Noto font has all of the accented characters. But let's say Blinka happened to be from Greece and had Greek characters. CircuitPython will handle this with no problems.



That's because CircuitPython handles unicode. In this case since Blinka isn't normally composed of Greek characters, we chose some equivalent Greek characters to demonstrate the badge's ability to display Greek. But what about Cyrillic characters like Russian? Yes, no problem.

Other Character Sets

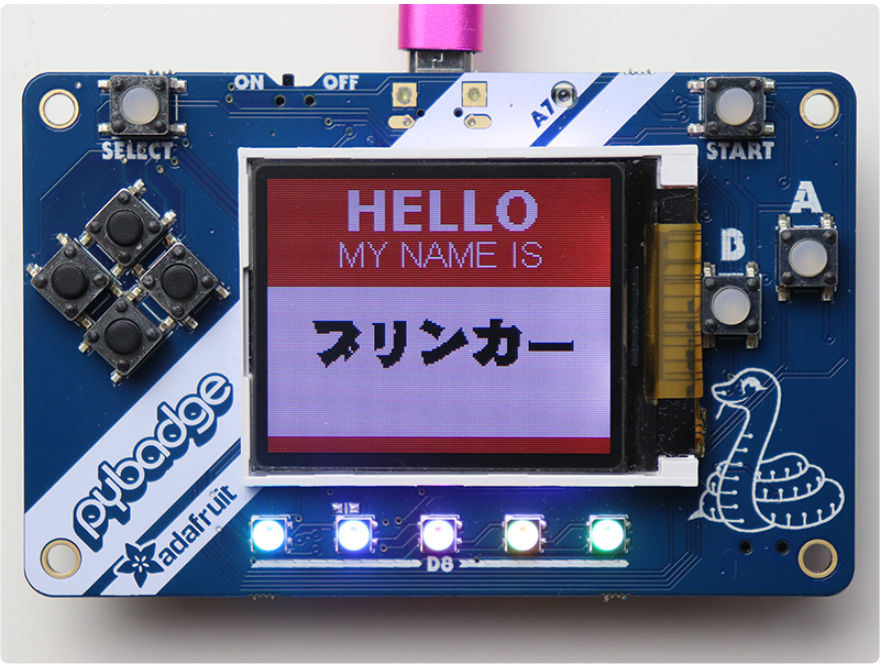
Showing characters in non-latin based languages can be a bit more challenging. This is due to the larger file size of the font and potential font mapping issues.

Ok, so we know we can write out the name in many languages just with these three character sets. If you have a name that uses another character set such as Arabic, Thai, Korean, Japanese or Chinese, there are plenty more font sets available from Google. In fact they have a page where you can type in your language and it will show you the fonts they have available in that language.

Google Noto Fonts

https://adafru.it/ETV

Of course you aren't limited to these fonts. You may find other font sets are much smaller and fit into the memory space on the PyBadge better. Here's an example of a Japanese Font with only katakana characters.



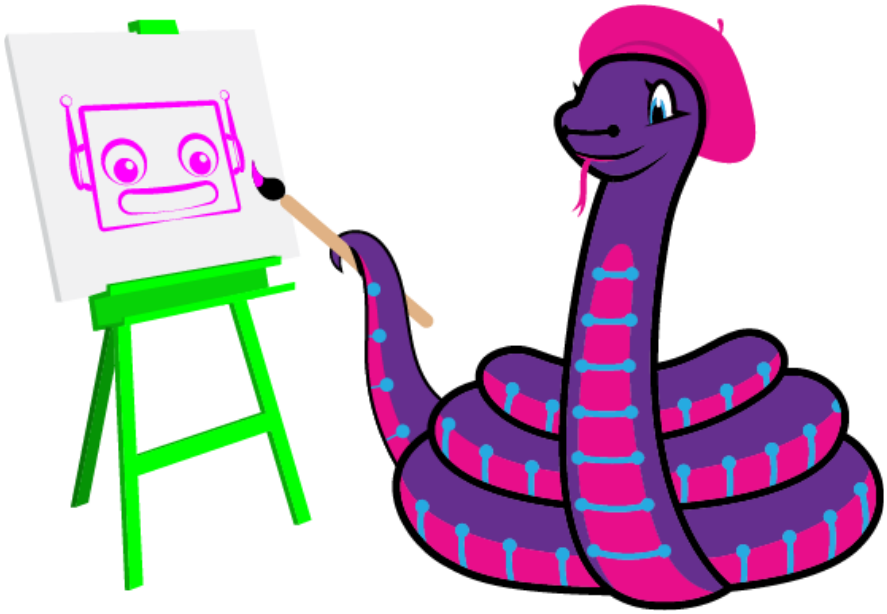
CircuitPython is even capable of handling fonts that should display Right to Left, with a small caveat. You will need to reverse the string before displaying it. You can easily do this with the following line of code.

```
NAME_STRING = ''.join(reversed(NAME_STRING))
```

Afterwards, it should display similar to this name tag that displays Blinka in Hebrew.

Changing the Hello Message

What if you would like to change the Hello message on the Badge? Yes, you can easily do that. Let's say we have a French Blinka.



We can easily change the labels to say the equivalent message in French.

Now let's take a look at some other customizations we can do to the badge on the following pages.

Customizing the Name Tag

The name tag can be customized in a number of ways. Let's take a look at the easiest option to change first.

Basic Customizations

```
HELLO_STRING = "HELLO"
MY_NAME_STRING = "MY NAME IS"
NAME_STRING = "Blinka"
NAME_FONTNAME = "/fonts/Noto-18.bdf"
NEOPIXEL_COUNT = 5
BACKGROUND_COLOR = 0xFF0000
FOREGROUND_COLOR = 0xFFFFFF
BACKGROUND_TEXT_COLOR = 0xFFFFFF
FOREGROUND_TEXT_COLOR = 0x000000
```

The most obvious customization is changing your name. Just edit `NAME_STRING`. You can even us a `\n` newline character in the middle of your string to break your information into two lines. The text will automatically center both horizontally and vertically. You can also change the message at the top by editing `HELLO_STRING` and `MY_NAME_STRING`.

The next items to change are the colors of the badge. Colors are Hexadecimal values in the format of RRGGBB. For instance, you can change the red `BACKGROUND_COLOR`, which is currently 0xFF0000, to anything you like as well as the white `FOREGROUND_COLOR`.

For instance, if we set `BACKGROUND_COLOR` to 0x0000FF, which is Blue and `FOREGROUND_COLOR` to 0xFFFF00, which is yellow, the result would look something like this.

You can also easily set the `BACKGROUND_TEXT_COLOR` to change the text on top of the red background or the `FOREGROUND_TEXT_COLOR` to change the color of your name.

If you have a custom font that you would like to use, you can set it by changing the `NAME_FONTNAME` to correct path.

The last item here that you can change is the `NEOPIXEL_COUNT`. If you have one of the PyBadge LC models with only one NeoPixel, you will want to change this to `1`.

Advanced Customizations

Additionally, you can change anything else you want in this name tag which includes:

- The fonts of the other labels
- Positions of all the labels
- Individual colors of the labels
- Adding graphics
- Storing and loading the current settings from non-volatile memory
- Anything else you can think of

Using the NeoPixels

Since the badge has built-in NeoPixels, let's animate them.

Now let's take a look at the code that does the animation.

CircuitPython NeoPixel Code

```
brightness = 0.2
direction = 1
speed = 1

# Define the NeoPixel
neopixels = neopixel.NeoPixel(
    board.NEOPIXEL,
    NEOPIXEL_COUNT,
    brightness=settings["brightness"],
    auto_write=False,
    pixel_order=neopixel.GRB,
)

# Remap the calculated rotation to 0 - 255
def remap(vector):
    return int(((255 * vector + 85) * 0.75) + 0.5)

# Calculate the Hue rotation starting with Red as 0 degrees
def rotate(degrees):
    cosA = cos(radians(degrees))
    sinA = sin(radians(degrees))
    red = cosA + (1.0 - cosA) / 3.0
    green = 1./3. * (1.0 - cosA) + sqrt(1./3.) * sinA
    blue = 1./3. * (1.0 - cosA) - sqrt(1./3.) * sinA
    return (remap(red), remap(green), remap(blue))

palette = []
pixels = []

# Generate a rainbow palette
for degree in range(0, 360):
    color = rotate(degree)
    palette.append(color[0] &lt;&lt; 16 | color[1] &lt;&lt; 8 | color[2])

# Create the Pattern
for x in range(0, NEOPIXEL_COUNT):
```

```
    pixels.append(x * 360 // NEOPIXEL_COUNT)

while True:
    for color in range(0, 360, speed):
        for index in range(0, NEOPIXEL_COUNT):
            palette_index = pixels[index] + color * direction
            if palette_index &gt;= 360:
                palette_index -= 360
            elif palette_index &lt; 0:
                palette_index += 360
            neopixels[index] = palette[palette_index]
        neopixels.show()
        neopixels.brightness = brightness
```

This can be broken down. Let's look at each section.

We will begin by including `NEOPIXEL_COUNT`, which was discussed in the previous section. Next, we set the default brightness, which is a `float` value between 0.0 and 1.0, the direction which will be either 1 or -1, and the speed which has to be a positive `int` or integer. Finally, we define the NeoPixel object.

```
NEOPIXEL_COUNT = 5

brightness = 0.2
direction = 1
speed = 1

# Define the NeoPixel
neopixels = neopixel.NeoPixel(
    board.NEOPIXEL,
    NEOPIXEL_COUNT,
    brightness=settings["brightness"],
    auto_write=False,
    pixel_order=neopixel.GRB,
)
```

In the next section we define a couple of functions that do the math for generating the rainbow palette. The `rotate()` function does most of the work and takes in a number between 0 and 360 degrees, with 0 degrees being red, and performs calculations to get the red, blue, and green component values. The values are then remapped with the `remap()` function to be between 0 and 255.

```
# Remap the calculated rotation to 0 - 255
def remap(vector):
    return int(((255 * vector + 85) * 0.75) + 0.5)

# Calculate the Hue rotation starting with Red as 0 degrees
def rotate(degrees):
    cosA = cos(radians(degrees))
    sinA = sin(radians(degrees))
    red = cosA + (1.0 - cosA) / 3.0
    green = 1./3. * (1.0 - cosA) + sqrt(1./3.) * sinA
    blue = 1./3. * (1.0 - cosA) - sqrt(1./3.) * sinA
    return (remap(red), remap(green), remap(blue))
```

In the next section, we pre-calculate the red, blue, and green components of each of the colors so that we don't need to do as much math during the animation sequence and add them into a palette array. For the pattern, we are really just trying to choose some numbers that are distributed evenly.

```
palette = []
pixels = []

# Generate a rainbow palette
for degree in range(0, 360):
    color = rotate(degree)
    palette.append(color[0] &lt;&lt; 16 | color[1] &lt;&lt; 8 | color[2])

# Create the Pattern
for x in range(0, NEOPIXEL_COUNT):
    pixels.append(x * 360 // NEOPIXEL_COUNT)
```

Finally, we get to the main loop and do a simple palette rotation. We do this by going through all of the degrees in the color wheel. Depending on the speed, we may skip a few steps to make it go faster. We simply need to get the index of the color we want in the palette that we had pre-calculated.

We can change the direction by making the index a negative number, which is then adjusted to be within the range of 0-360 and we have our index. We look up the color by providing the index to the palette and then assign the color to the neopixel.

We also set the brightness of the NeoPixels here in case we changed it.

```
while True:
    for color in range(0, 360, speed):
        for index in range(0, NEOPIXEL_COUNT):
            palette_index = pixels[index] + color * direction
            if palette_index &gt;= 360:
                palette_index -= 360
            elif palette_index &lt; 0:
                palette_index += 360
            neopixels[index] = palette[palette_index]
        neopixels.show()
        neopixels.brightness = brightness
```

PyBadge Buttons

To use the PyBadge buttons, we will need a few pieces of code. Here's how the buttons function:

- Pushing **Up** make the lights cycle faster
- Pushing **Down** makes the lights cycle slower
- Pushing **Right** makes the lights appear to move to the right
- Pushing **Left** makes the buttons appear to move to the left

- Pushing the **B Button** makes the lights dimmer
- Pushing the **A Button** makes the light brighter

CircuitPython Button Code

```
BUTTON_LEFT = const(7)
BUTTON_UP = const(6)
BUTTON_DOWN = const(5)
BUTTON_RIGHT = const(4)
BUTTON_SEL = const(3)
BUTTON_START = const(2)
BUTTON_A = const(1)
BUTTON_B = const(0)

# Define Events and Shift Register
latest_event = Event()
last_event = Event()

pad = ShiftRegisterKeys(clock=board.BUTTON_CLOCK,
                    data=board.BUTTON_OUT,
                    latch=board.BUTTON_LATCH,
                    key_count=8,
                    value_when_pressed=True,
                    interval=0.1,
                    max_events=1)

def check_buttons(event):
    if event.key_number == BUTTON_RIGHT:
        settings["direction"] = -1
    elif event.key_number == BUTTON_LEFT:
        settings["direction"] = 1
    elif (event.key_number == BUTTON_UP) and settings["speed"] &lt; 10:
        settings["speed"] += 1
    elif (event.key_number == BUTTON_DOWN) and settings["speed"] &gt; 1:
        settings["speed"] -= 1
    elif (event.key_number == BUTTON_A) and settings["brightness"] &lt; 0.5:
        settings["brightness"] += 0.025
    elif (event.key_number == BUTTON_B) and settings["brightness"] &gt; 0.025:
        settings["brightness"] -= 0.025

current_buttons = pad.get_pressed()
last_read = 0
while True:
    for color in range(0, 360, speed):
        pad.events.get_into(latest_event)
        if latest_event.pressed and latest_event.key_number !=
last_event.key_number:
            check_buttons(latest_event)
            last_event = latest_event
            latest_event = Event(
                key_number=8
            )  # An imaginary key number that doesn't exist!
```

This code uses ShiftRegisterKeys, which allows reading the state of the buttons from the internal shift register. This means we need to detect when a button is pressed and released. We do this by monitoring the state of all the buttons and respond only when there is a change. Let's look at the code a little closer.

```
BUTTON_LEFT = const(7)
BUTTON_UP = const(6)
BUTTON_DOWN = const(5)
BUTTON_RIGHT = const(4)
BUTTON_SEL = const(3)
BUTTON_START = const(2)
BUTTON_A = const(1)
BUTTON_B = const(0)
```

There are eight buttons on the PyBadge and each corresponds to the position of a bit in the byte of data that is returned. In the first part, we define which button corresponds to the appropriate value. We do that by defining each of the buttons as bit masks. Bit masking works by defining which bits we want to look at and which we want to hide or "mask". For instance we have `BUTTON_SEL` defined as 3, which is bit 3 in an 8-bit array, so it looks for `0x00001000` in binary.

If ShiftRegisterKeys gets back a value that is something like `0x01001000` then it knows that the `BUTTON_SEL` is being pressed at the same time as `BUTTON_UP` which is bit 7 or `0x01000000`.

In the next segment of code we create a couple of `Event` objects and setup the `ShiftRegisterKeys` object and call it **pad**.

```
latest_event = Event()
last_event = Event()

pad = ShiftRegisterKeys(
    clock=board.BUTTON_CLOCK,
    data=board.BUTTON_OUT,
    latch=board.BUTTON_LATCH,
    key_count=8,
    value_when_pressed=True,
    interval=0.1,
    max_events=1,
)
```

We do that by passing the `Clock`, `Latch`, and `Out` pins of the built-in shift register to the module. This is an SN74HC165 Parallel-Load Shift Register, which works by first "latching" the current input values of the buttons being pressed. Then the `Out` pin is read and the data is shifted each time a Clock pulse is sent. This way we can get any combination of the eight inputs from the shift register. This is all handled in the background which makes it much easier to use.

The `interval` of **0.1**, which is expressed in seconds, is important because it tells how CircuitPython how long to wait before reading the shift register. If it is read too frequently, then sometimes incorrect values are returned.

Next is the `check_buttons()` function. This function will read the event that is passed in to determine which buttons are being pressed and update the settings

dictionary. By putting the checks in an else/if structure, it forces the code to only respond to one button press at a time.

```
def check_buttons(event):
    if event.key_number == BUTTON_RIGHT:
        settings["direction"] = -1
    elif event.key_number == BUTTON_LEFT:
        settings["direction"] = 1
    elif (event.key_number == BUTTON_UP) and settings["speed"] &lt; 10:
        settings["speed"] += 1
    elif (event.key_number == BUTTON_DOWN) and settings["speed"] &gt; 1:
        settings["speed"] -= 1
    elif (event.key_number == BUTTON_A) and settings["brightness"] &lt; 0.5:
        settings["brightness"] += 0.025
    elif (event.key_number == BUTTON_B) and settings["brightness"] &gt; 0.025:
        settings["brightness"] -= 0.025
```

Finally let's take a look at the button code in the main loop, which handles the button presses.

```
current_buttons = pad.get_pressed()
last_read = 0
while True:
    for color in range(0, 360, speed):
        pad.events.get_into(latest_event)
        if latest_event.pressed and latest_event.key_number !=
last_event.key_number:
            check_buttons(latest_event)
            last_event = latest_event
            latest_event = Event(
                key_number=8
            )  # An imaginary key number that doesn't exist!
```

We start by grabbing the current value of the buttons and setting the time of the last read to 0. After that, we have a `for` loop to cycle through all of the different colors. I have omitted that code so that we can focus on the button code only.

In the last bit, it waits until the button is no longer pressed so that the button press events aren't continuously fired off, which would make changing settings very difficult.