Hacking Ikea Lamps with Circuit Playground Express

Created by Kattni Rembor

https://learn.adafruit.com/hacking-ikea-lamps-with-circuit-playground-express

Last updated on 2023-08-29 03:40:03 PM EDT

Table of Contents

Lamp it up!

Do you have an old lamp laying around? Maybe your favorite lamp could use an upgrade! In this guide, you will learn how to take your Circuit Playground Express and use Circuit Python to make an interactive, colorful lamp. We've chosen two Ikea lamps, SPOKA and SJOPENNA, for this project. However, this concept will work with any lamp that you can fit a Circuit Playground Express into and connect it to USB power.

We wanted this project to be as interactive as possible, and that requires some special CircuitPython code. This guide is packed with really cool coding concepts, including a little bit of math. You'll find out different ways to use time. You'll learn how to code different interactions to work together by using generators. You'll discover how to keep repetitive code efficient by using a dictionary. Finally, you'll make an input require multiple steps by creating a state machine.

Let's get started!

First Things First

The first thing you'll want to do is get your Circuit Playground Express and your computer ready to follow along in the guide. If you're completely new to CircuitPython, checkout the Welcome to CircuitPython guide () before continuing.

You'll want to make sure your CPX is updated to the most recent version of CircuitPython. Take a look at Installing CircuitPython () to get updated if you need to.

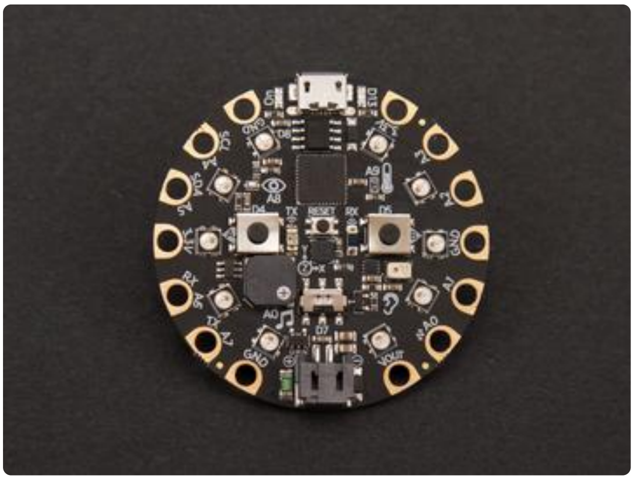
CircuitPython code relies on libraries to function. The code in this project will not work without the correct libraries installed! Take a look at CircuitPython Libraries () to learn about Installing the CircuitPython Library Bundle ().

You'll be writing and editing code through this guide. So, you'll want to choose an editor. You can use any plain-text editor, but code editors make everything much simpler. There are many options. We recommend Mu editor as it is a code editor and serial console all in one! Check out Installing Mu Editor () to get that going. You may want to check out Creating and Editing Code () if you haven't worked with Mu or CircuitPython code before.

You'll be making significant use of the serial console and REPL. Before getting started, you'll want to make sure you have your serial connection setup. If you're using Mu Editor, you're all set to go already! Check out Connecting to the Serial Console () to see how to use the REPL within Mu Editor. If you're using a different editor, you'll want to check out Advanced Serial Console on Windows () and Advanced Serial Console on Mac and Linux () to get connected. Once connected, take a look at Interacting with the Serial Console () and The REPL () to get a feel for both.

This guide assumes that you've completed these steps. If you reach a point in the guide where you realise you may have missed one, refer back to this page to get things taken care of and then continue.

Things You'll Need For This Project



Circuit Playground Express

USB to Micro USB cable

Mini Remote Control

Any lamp that fits the Circuit Playground Express

Some form of adhesive to attach the Circuit Playground Express inside your lamp, such as:

Double-sided foam tape

Sugru

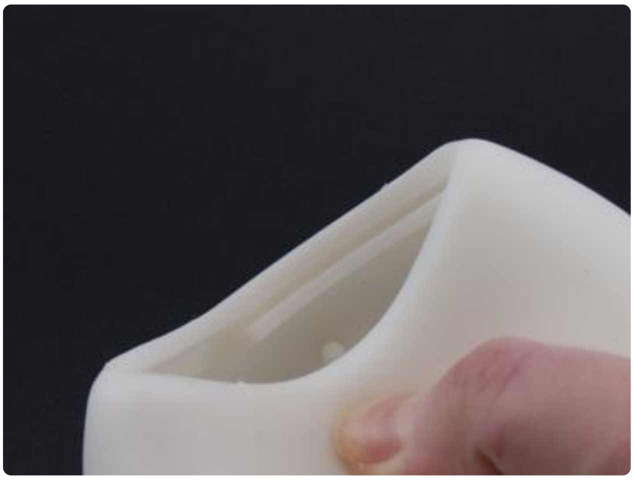
Fitting Circuit Playground Express

We'll be showing off two mods with lamps we got at our local IKEA but you don't have to use these lamps. They work fantastically but other lamps you may already own could also work well. Also, check out your local thrift/antique store for funky finds.

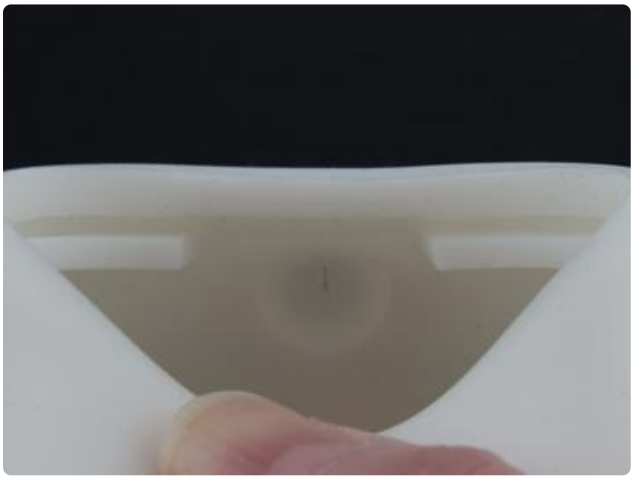
SPOKA Lamp

Our little creature friend Spoka () seems to have been made for Circuit Playground Express. The outside casing is stretchy silicone. The inside is a hard plastic self-contained housing with the electronics for the lamp. If your lamp is plugged in, unplug it first. Then, follow the steps below.

Stretch the outside casing off of the internal plastic housing.



Once separated, you'll notice a groove around the inside of the round bottom opening. In the back, there is a port for charging, and a space that's perfect for the USB port on the CPX to have enough clearance to be plugged in.



Grab your USB micro cable and run it through the charging port in the back. Plug it into your CPX before you fit the CPX into the lamp - it's difficult to do it once it's fitted in!

Flip the CPX so the NeoPixels are pointing towards the inside of the lamp.

Press the CPX into the groove, starting with the section near the USB. Apply a constant gentle pressure to keep the CPX in the groove as you go. Gently stretch the lamp housing around the CPX until it's entirely fitted into the bottom of the lamp.

The Circuit Playground Express fits perfectly!

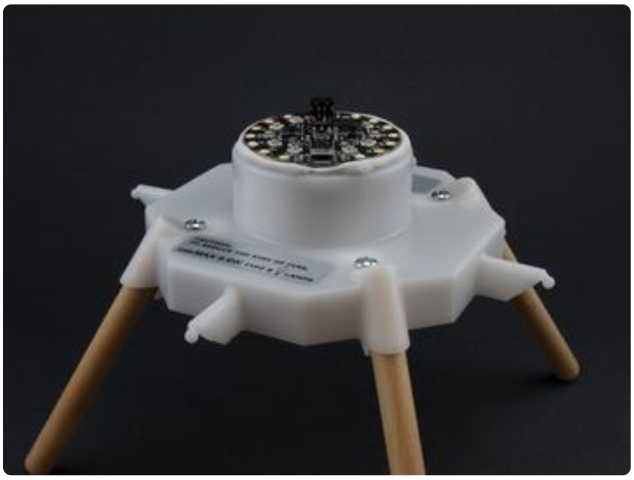
SJOPENNA Lamp

The raised part of the base of the geometrical Sjopenna () is a perfect place to attach your Circuit Playground Express.

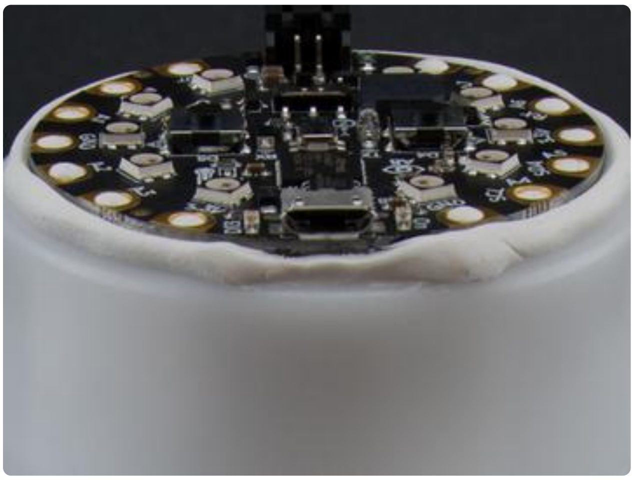
You'll need some kind of adhesive to attach it as there is nothing to fit it into. Simple double-sided tape wasn't thick enough or sticky enough. Double-sided sponge tape worked very well.

If you want something more permanent or simply feel like getting fancy, consider using Sugru (), the awesome mouldable glue!

Sugru It To It!



Open up the packet, take the Sugru out and make it into a snake the right length to circle around the Circuit Playground Express. Place it around the back of the CPX, and then press it down onto the base of the lamp.



Use a spudger, finger nail, or some other tool to make an indentation in the Sugru near the micro USB port to ensure that any USB cable you use to power your CPX has room to plug in.

Follow the instructions to let it sit, and you're all set to go!

Blink vs Blink

Every programmer in every language, sometime early in their learning, has written a program called `"Hello world!"` that prints exactly that. The idea behind it is it's an excellent introduction to the language and programming environment. In CircuitPython, our `Hello World!` is also known as Blink. Every CircuitPython compatible board has a little red LED on it, and you can use a simple program to make it blink. After all, making things blink is great!

Here is what the basic Blink looks like on the Circuit Playground Express. Try loading it on your board to see what happens!

```
# SPDX-FileCopyrightText: 2017 Kattni Rembor for Adafruit Industries
#
# SPDX-License-Identifier: MIT

import time

from adafruit_circuitplayground.express import cpx

while True:
    cpx.red_led = not cpx.red_led
    time.sleep(0.5)
```

The little red LED is `True` when it's on, and `False` when it's off. This code cycles back and forth between the LED being on for 0.5 seconds and off for 0.5 seconds.

For this project, we will be using the built in NeoPixels. So, for this example, we've written a version of Blink that uses the first NeoPixel instead. However, setting a NeoPixel to a color is not a `True` or `False` situation. We're setting it to a color which is done using a tuple in `(Red, Green, Blue)` format. Cycling back and forth here involves cycling between red which is `(255, 0, 0)` and off which is `(0, 0, 0)`. So, to do this, we're going to need to do a little math.

Absolute Value

The absolute value of a number can be thought of as that number's distance from zero. For example, the absolute value of 255 is 255. The absolute value of -255 is also 255, because even though it's a negative number, it is still 255 away from zero. To obtain the absolute value of a number using CircuitPython, you use `abs()`. You can get the absolute value of a single number or you can get the absolute value result of an equation. Try typing different numbers into your REPL to see the results. For example:

```
Adafruit CircuitPython 2.2.3 on 2018-02-05; Adafruit CircuitPlayground Express
with samd21g18
>>> abs(255)
255
>>> abs(-255)
255
>>> abs(255 - 255)
0
>>> abs(0 - 255)
255
>>>
```

`-255` and `0 - 255` are both equal to `-255`, however, as you can see, the absolute value of both is `255`.

What does this have to do with blinking our NeoPixel? We're going to use `abs()` in our Blink code to cycle between 255 and 0. This will allow us to cycle back and forth between red and off. Since we're only changing one number in the `(R, G, B)` tuple, our NeoPixel Blink code looks like this:

```
# SPDX-FileCopyrightText: 2017 Kattni Rembor for Adafruit Industries
#
# SPDX-License-Identifier: MIT

import time

from adafruit_circuitplayground.express import cpx

while True:
    cpx.pixels[0] = (abs(cpx.pixels[0][0] - 255), 0, 0)
    time.sleep(0.5)
```

This cycles between `cpx.pixels[0] = (255, 0, 0)` and `cpx.pixels[0] = (0, 0, 0)`. Load it on your Circuit Playground Express and see the difference!

Next, we're going to use this example to learn a new concept!

Passing Time

There are many situations where you'll want an event in your code to continue for an amount of time. Often, this is accomplished using `time.sleep()` as in the following code:

```
# SPDX-FileCopyrightText: 2017 Kattni Rembor for Adafruit Industries
#
# SPDX-License-Identifier: MIT

import time

from adafruit_circuitplayground.express import cpx

while True:
    cpx.pixels[0] = (abs(cpx.pixels[0][0] - 255), 0, 0)
    time.sleep(0.5)
```

Here, the first NeoPixel turns on for 0.5 seconds, and then turns off for 0.5 seconds before repeating indefinitely. The usage of `time.sleep(0.5)` in this code basically says: turn the LED on and wait in that state for half a second, then turn it off and wait in that state for half a second. In many situations, this usage of `time` works great. However, during `time.sleep()`, the code is essentially paused. Therefore, the board cannot accept any other inputs or perform any other functions for that period of time. This type of code is referred to as being blocking. In the case of the code above, this is sufficient as the code is not attempting to do anything else during that time.

Waiting Without Blocking

However, for this project, we want to continue processing inputs, so instead of `sleep`ing for 0.5 seconds, we'll process other inputs for 0.5 seconds and change the led when that time expires. To accomplish this, we're going to use `time.monotonic()`. Where `time.sleep()` expects an amount of time be provided, `time.monotonic()` tells us what time it is now, so we can see whether our `0.5` seconds has passed yet. So, we no longer supply an amount of time. Instead, we assign `time.monotonic()` to two different variables at two different points in the code, and then compare the results.

At any given point in time, `time.monotonic()` is equal to the number seconds since your board was last power-cycled. (The soft-reboot that occurs with the auto-reload when you save changes to your CircuitPython code, or enter and exit the REPL, does not start it over.) When it is called, it returns a number with a decimal, which is called a `float`. If, for example, you assign `time.monotonic()` to a variable, and then call it again to assign into a different variable, each variable is equal to the number of seconds that `time.monotonic()` was equal to at the time the variables were assigned. You can then subtract the first variable from the second to obtain the amount of time that passed.

`time.monotonic()` example

Let's take a look at an example. You can type the following into the REPL to follow along.

```
Adafruit CircuitPython 2.2.1 on 2018-01-23; Adafruit CircuitPlayground Express with samd21g18
>>> import time
>>> print(time.monotonic())
769138.0
>>> x = time.monotonic()
>>> y = time.monotonic()
>>> print(y - x)
8.5
>>> print(time.monotonic())
769165.0
>>>
```

First we `import` the time module, then we `print` the current `time.monotonic()`. This is to give you an idea of what is going on in the background. The next two lines assign `x = time.monotonic()` and `y = time.monotonic()` so we have two variables, and points in time, to compare. Then we `print(y - x)`. This gives us the amount of time, in seconds, that passed between assigning `time.monotonic()` to `x` and `y`. We `print time.monotonic()` again to give you a general idea of the difference. Remember, the two numbers resulting from printing the current time are not exactly the same difference from each other as the two variables due to the amount of time it took to assign the variables and print the results.

Non-Blocking Blink

But, how does this allow us to blink our NeoPixel? The result of the comparison is a period of time. So, if we use that period of time to determine when the state of the LED should change, we can successfully blink the LED in the same way we did in the first program. Let's find out what that looks like!

```
# SPDX-FileCopyrightText: 2018 Kattni Rembor for Adafruit Industries
#
# SPDX-License-Identifier: MIT

import time

from adafruit_circuitplayground.express import cpx

blink_speed = 0.5

cpx.pixels[0] = (0, 0, 0)

initial_time = time.monotonic()
while True:
    current_time = time.monotonic()
    if current_time - initial_time > blink_speed:
        initial_time = current_time
        cpx.pixels[0] = (abs(cpx.pixels[0][0] - 255), 0, 0)
```

This does exactly the same thing as before! It's exactly what we wanted. Now, let's break it down.

Before the loop begins, we create a `blink_speed` variable and set it to `0.5`. This allows for easier configuration of the blink speed later if you wanted to alter it. Next, we set the initial state of the LED to be `(0, 0, 0)`, or off. Then, we call `time.monotonic()` for the first time by setting `initial_time = time.monotonic()`. This applies once when the program begins, before it enters the loop.

Once the code enters the loop, we set `current_time = time.monotonic()`. We call it a second time to compare to the first, to see if enough time has passed. Then we say if `current_time` minus `initial_time` is greater than `blink_speed`, do two things: set `initial_time` to now be equal to `current_time` and cycle the NeoPixel to the next state. Setting `initial_time = current_time` means it starts the time period over again. Essentially, every time the difference reaches `0.5` seconds, it cycles the state and starts again, repeating indefinitely.

Why would we do it this way? It seems way more complicated! We do it this way because this allows us to do other things while the NeoPixel is blinking. Instead of pausing the code to leave the LED in a red or off state, the code continues to run. The code for the Spoka lamp allows you to change speed and brightness without halting the rainbow animation, and this is how we accomplish that!

Dictionary or Else

Let's keep adding tools that we can use to make this lamp really awesome. Non-blocking waiting is going to be helpful. Another useful concept we'll lean on later is using python dictionaries.

You will often find yourself writing code that seems repetitive. You may have a number of lines of code that are a series of similar elements using `if` and `elif` statements. Let's take a look at an example.

```
# SPDX-FileCopyrightText: 2018 Kattni Rembor for Adafruit Industries
#
# SPDX-License-Identifier: MIT

from adafruit_circuitplayground.express import cpx

while True:
    if cpx.touch_A1:
        cpx.pixels.fill((255, 0, 0))  # red
    elif cpx.touch_A2:
        cpx.pixels.fill((255, 40, 0))  # orange
    elif cpx.touch_A3:
        cpx.pixels.fill((255, 150, 0))  # yellow
    elif cpx.touch_A4:
        cpx.pixels.fill((0, 255, 0))  # green
    elif cpx.touch_A5:
        cpx.pixels.fill((0, 0, 255))  # blue
    elif cpx.touch_A6:
        cpx.pixels.fill((180, 0, 255))  # purple
    elif cpx.touch_A7:
        cpx.pixels.fill((0, 0, 0))  # off
```

In this example, we are changing the colors of the NeoPixels using the capacitive touch pads. Each touch pad changes it to a different color, and the last touch pad turns them off.

This code seems really repetitive though. The difference between the `if` and `elif` statements is which touch pad, and the difference between the color statements is the color. Is there another way to do this? Yes! We're going to use a dictionary.

The CircuitPython Dictionary

A dictionary maps a set of objects, called `keys`, to another set of objects, called `values`. These are expressed in `key: value` pairs. Within a single dictionary, each `key` must be unique. A dictionary can contain all different types of information. Dictionaries have a name and the information is contained within `{}` brackets.

For our dictionary, we're going to pair a string with a tuple. Our string will be the names of the different touch pads, and the tuple will be the associated `(r, g, b)` color value. Let's take a look what our code looks like with a dictionary for our data.

```
# SPDX-FileCopyrightText: 2018 Kattni Rembor for Adafruit Industries
#
# SPDX-License-Identifier: MIT

from adafruit_circuitplayground.express import cpx

touchpad_to_color = {
    "touch_A1": (255, 0, 0),  # red
    "touch_A2": (255, 40, 0),  # orange
    "touch_A3": (255, 150, 0),  # yellow
    "touch_A4": (0, 255, 0),  # green
    "touch_A5": (0, 0, 255),  # blue
    "touch_A6": (180, 0, 255),  # purple
    "touch_A7": (0, 0, 0),  # off
}

while True:
    for touchpad in touchpad_to_color:
        if getattr(cpx, touchpad):
            cpx.pixels.fill(touchpad_to_color[touchpad])
```

We've named our dictionary `touchpad_to_color`. Our `keys` are strings with the touch pad names. Our `values` are the colors we chose to associate with each touch pad. For example, the first `key: value` pair is `"touch_A1": (255, 0, 0)`. Note that the `"touch_A1"` including the quotation marks is the string, and as a whole is the `key`. Now that we have our dictionary, we can use it in our code.

Our loop starts with `for touchpad in touchpad_to_color:`. This line ensures that the code only applies if the `key` is found within the dictionary. Otherwise, you'll get a `KeyError` if you try to apply a key that isn't found! This check avoids that. The key thing to know is that `for key in dictionary` steps over all the `keys`. So not only does it ensure the key is found, it ensures the entire dictionary is utilised. Then with our `if` statement we say, `if` a `key` touch pad has been touched, use the paired `value` to `fill` the NeoPixels the associated color.

We went from seven `if` and `elif` statements to one if statement. We add some data but we save a lot of code in our loop. Data is easier to manipulate than code. So, in the event we wanted to add more touch pads to our dictionary, we add only one line of data into our dictionary for every two lines of code we would have had to add without it!

Generate Your Colors

Our little creature friend is a great opportunity for rainbow lighting. The rainbow code is often referred to as "rainbow_cycle ()" for a reason: it is exactly that, a cycle that starts with red, then orange, yellow, green, blue, violet and back to red.

Normally, this cycle must complete before the board can continue to look for inputs. At this point, you must either pause the cycle to wait for input or time the input to happen exactly between cycles. Basically, the rainbow_cycle is blocking - just like we saw with `time.sleep()`.

This won't work for us! Why? Because rainbows are great! We don't want them to stop while we're doing other things. We want to be able to change the brightness and speed of the rainbow without waiting for the cycle to complete.

Generators to the rescue!

To do this, we're going to use something called a generator. A generator function contains a `yield` statement. You can call `next` on a generator and with every call, it returns a value until it has returned all possible values. The great thing about generators is that they save the state when they `yield` so we can get right back to where we were, as though we never left. This is important for us because we want our cycle to continue while we process other things.

In this example, we're going to combine `time.monotonic()` and dictionaries with the new generators that we're about to learn!

But first, a quick explanation of how we're getting our colors.

`colorwheel` (or `wheel`) Explained

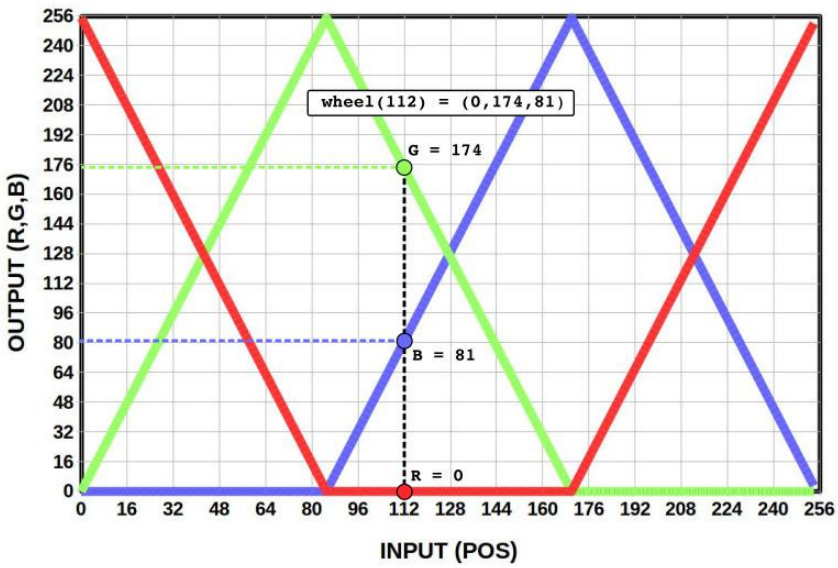
You have probably come across this code in a number of situations. It's in much of the code that has a rainbow cycle option, and is included in the demos that ship on many of the CircuitPython compatible boards. But how does it work?

```
def wheel(pos):
    # Input a value 0 to 255 to get a color value.
    # The colours are a transition r - g - b - back to r.
    if pos &lt; 0 or pos &gt; 255:
```

```
        return 0, 0, 0
    if pos &lt; 85:
        return int(255 - pos*3), int(pos*3), 0
    if pos &lt; 170:
        pos -= 85
        return 0, int(255 - pos*3), int(pos*3)
    pos -= 170
    return int(pos * 3), 0, int(255 - (pos*3))
```

The `wheel` code is a function that uses math to allow a single number to represent the `(r, g, b)` tuple that usually represents pixel colors. If you wanted to turn your LEDs red, you'd usually use `cpx.pixels.fill((255, 0, 0))`. However, with `wheel`, if you include the function at the top of your program, you can use `cpx.pixels.fill(wheel(0))`.

Here is what `wheel` looks like graphed out. The x-axis is the number you provide, and the y-axis indicates, based on the color lines, what tuple will be returned. As you can see, if you provide `wheel(112)`, it returns the `(R, G, B)` tuple `(0, 174, 81)`.



Now, if all you're doing is using solid colors, it doesn't make much sense to use `wheel`, because it adds a lot to your code. However, if you want to do a rainbow cycle, `wheel` is the answer. The typical rainbow cycle uses fancy math code to give `wheel` a sequence of numbers from `0` to `255`, to iterate through all the possible colors from red, to green to blue, and back to red again. The rainbow cycle code is designed to continuously do this. So, even though it's only displaying a single color at any given point in time, when it's viewed altogether, it appears to be a beautiful rainbow!

This is important to know because, in our generator code, we're going to use `wheel` to create our rainbow cycle mode, but we're also going to use it to create our individual single color modes. Now that we understand how `wheel` works, the list we use for our color mode sequence generator will make a lot more sense!

The `next` Step

```
# SPDX-FileCopyrightText: 2018 Kattni Rembor for Adafruit Industries
#
# SPDX-License-Identifier: MIT

import time
from rainbowio import colorwheel
from adafruit_circuitplayground.express import cpx


# pylint: disable=stop-iteration-return
def cycle_sequence(seq):
    while True:
        for elem in seq:
            yield elem


def rainbow_lamp(seq):
    g = cycle_sequence(seq)
    while True:
        cpx.pixels.fill(colorwheel(next(g)))
        yield


color_sequences = cycle_sequence([
    range(256),  # rainbow_cycle
    [0],  # red
    [10],  # orange
    [30],  # yellow
    [85],  # green
    [137],  # cyan
    [170],  # blue
    [213],  # purple
    [0, 10, 30, 85, 137, 170, 213],  # party mode
])

heart_rates = cycle_sequence([0, 0.5, 1.0])

heart_rate = 0
last_heart_beat = time.monotonic()
next_heart_beat = last_heart_beat + heart_rate

rainbow = None

cpx.detect_taps = 2
cpx.pixels.brightness = 0.2

while True:
    now = time.monotonic()

    if cpx.tapped or rainbow is None:
        rainbow = rainbow_lamp(next(color_sequences))

    if cpx.shake(shake_threshold=20):
        heart_rate = next(heart_rates)
        last_heart_beat = now
        next_heart_beat = last_heart_beat + heart_rate

    if now >= next_heart_beat:
        next(rainbow)
        last_heart_beat = now
        next_heart_beat = last_heart_beat + heart_rate
```

Load the file on your CPX, and give it at try. It will start with a rainbow. If you double-tap your lamp, it will move to solid red. Double-tap once each to move to yellow, orange, green, cyan, blue and purple. Double-tap one more time to switch to party mode. In party mode, it's easiest to see the changes in speed. While in party mode, shake your lamp. The speed will slow down. Shake it again to slow it down even more. Shake it again, and it will speed up again.

Now let's find out how!

The Code!

We begin with `imports` and the `wheel` code.

Generators

First, we're going to first create a special generator, called `cycle_sequence`, that will allow our other generators to continuously cycle through their options.

```
def cycle_sequence(seq):
    while True:
        for elem in seq:
            yield elem
```

We do this because we're going to have different modes that we would like to repeatedly cycle through. For example, there are two rainbow settings and seven solid colors available for a total of nine color modes. Every call to a generator returns a value until all possible values have been generated. Without `cycle_sequence`, we would get through the nine color modes and the code would stop. With this special generator, the code will allow us to return to the first mode and start again. It's super useful!

Now we can use it to create our rainbow generator, `rainbow_lamp`.

```
def rainbow_lamp(seq):
    g = cycle_sequence(seq)
    while True:
        cpx.pixels.fill(wheel(next(g)))
        yield
```

It is different than the others. It expects to be provided with a sequence, instead of having one to iterate through on its own. `rainbow_lamp` uses `seq` from `cycle_sequence` to iterate through the sequence. The sequence we will provide it is contained within the next generator. We will use the next generator to provide the `(pos)` to `wheel` and create our different color modes.

The next two generators use `cycle_sequence` to iterate through a list of values. The first, `color_sequences`, is a list containing the different `(pos)` position values that will be provided to `wheel`.

```
color_sequences = cycle_sequence([
    range(256),  # rainbow_cycle
    [0],         # red
    [10],        # orange
    [30],        # yellow
    [85],        # green
    [137],       # cyan
    [170],       # blue
    [213],       # purple
    [0, 10, 30, 85, 137, 170, 213],  # party mode
])
```

The second generator, `heart_rates`, contains the speed of our modes in seconds.

```
heart_rates = cycle_sequence([0, 0.5, 1.0])
```

To be clear, this is not the speed to cycle between modes - that will be done with user input. This is the speed of the rainbow and party modes. Solid colors do not care about speed, so while the speed exists during those modes, it does not affect them.

Note that `color_sequences` and `heart_rates` are not functions like `rainbow_lamp`, however they are still generators because they use `cycle_sequence`.

Time

Remember, we learned that when nothing else is going on, we can use `time.sleep()` to control speed, however, if we want to be able to process anything else, we need to use `time.monotonic()`. In this code, we want to be able to process inputs while the rainbow cycle is happening. We will be able to change the speed of the rainbow while the rainbow is going, without halting or resetting the rainbow cycle!

The next section is where we setup what we're going to use with `time.monotonic()`.

```
heart_rate = 0
last_heart_beat = time.monotonic()
next_heart_beat = last_heart_beat + heart_rate
```

We learned that `time.monotonic()` is all about comparisons, so here we setup the variables we'll be comparing. We set `heart_rate = 0` for use later. Then we set `last_heart_beat = time.monotonic()` and `next_heart_beat = last_heart_beat + heart_rate`.

Variables

We need to assign a few more things before we get into our loop.

```
rainbow = None

cpx.detect_taps = 2
cpx.pixels.brightness = 0.2
```

First, we assign `rainbow = None` for later use. Then, we set `cpx.detect_taps = 2` so our code will use a double-tap for the `cpx.tapped` input. Last, we set `cpx.pixels.brightness = 0.2` so the brightness will be low on startup. This way if your CPX resets in the middle of the night, it doesn't come back on super bright!

The Loop

We begin by setting `now = time.monotonic()` to keep track of current time.

Our first `if` statement has two options.

```
    if cpx.tapped or rainbow is None:
        rainbow = rainbow_lamp(next(color_sequences))
```

One, we double-tap the lamp, and two, `rainbow is None`. If you recall, we assigned `rainbow = None` before the loop. So this statement is effectively saying, "If you double-tap or on startup, do the following." So, if either one of these options are met, we assign `rainbow = rainbow_lamp(next(color_sequences))`. This is where we begin using our generators and is the first time we call `next`! Remember, `rainbow_lamp` expects a sequence, and we are providing it exactly. Each time you double tap, it calls for the `next` value in `color_sequences`, which contains the different color modes. And because we're using our special generator, when we reach the last mode, another double-tap will cycle back to the first mode!

Next, we're using shake as the input to change speeds.

```
    if cpx.shake(shake_threshold=20):
        heart_rate = next(heart_rates)
```

```
        last_heart_beat = now
        next_heart_beat = last_heart_beat + heart_rate
```

Remember, the `heart_rates` generator provides the speeds. We assign `heart_rate` to call the `next` value in `heart_rates`. Then we use our `time.monotonic()` variables to check how much time has passed and set `next_heart_beat = last_heart_beat + heart_rate`. This is used in the last section of code to determine what speed is currently set and use it.

Our last section of code we are determining the speed at which we are calling `next` on `rainbow`. This is how we set the speed of each color mode. Remember, solid colors don't care about speed and simply aren't affected. This speed is important to the rainbow and party modes.

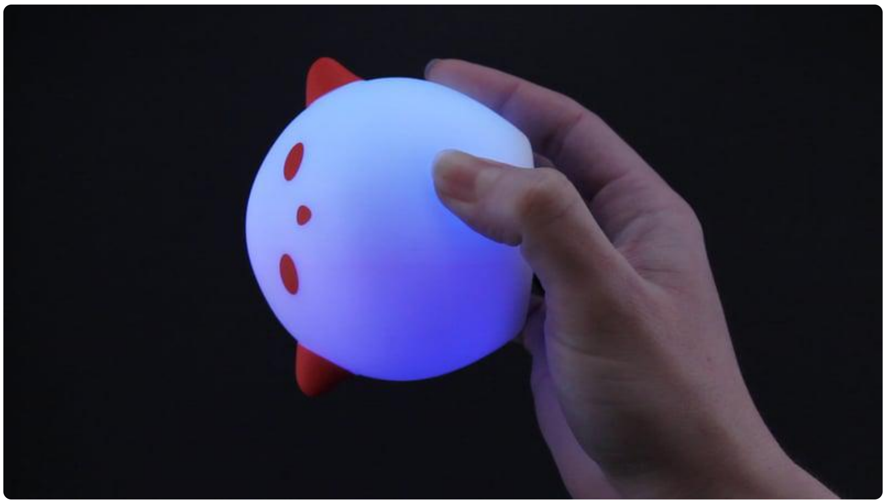
```
    if now &gt;= next_heart_beat:
        next(rainbow)
        last_heart_beat = now
        next_heart_beat = last_heart_beat + heart_rate
```

We check to see if `now` is greater than or equal to `next_heart_beat` (which we just set to be essentially `now + heart_rate`), and when it is, we call `next` on `rainbow`. This causes the rainbow cycle to move to the next `(pos)` in `wheel`. Lastly, we reset `last_heart_beat` and `next_heart_beat` so we can begin a new comparison, and continue on in our code!

Note:

Pylint ensures that code is written according to a particular standard. The `pylint` comments in the code are there because we chose not to follow the standard for part of our program, in order to keep the code as readable as possible. To learn more, checkout the [Pylint documentation ()]().

Rotation State



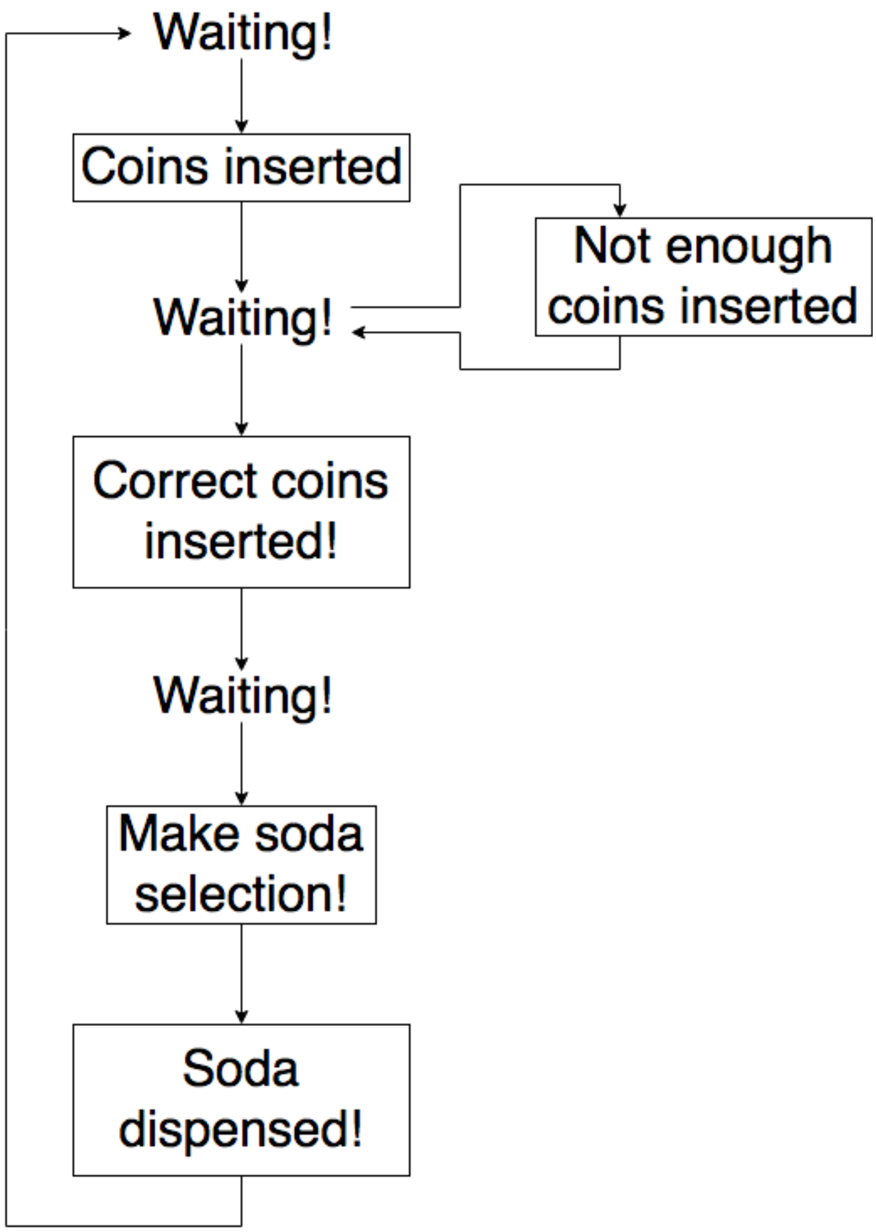
We want to use motion to change some settings on our little creature friend Spoka. In this case, we're going to use rotating the lamp to the left and to the right as two separate inputs. We plan to use shake as an input, so we need to make sure that our rotation motion isn't mistaken for a shake motion. There are various ways we could use orientation as a limiting factor, but many of them make using the actual input difficult and inconsistent. We want our lamp to work easily every time!

So, we're going to require a series of events to occur in a particular order for the input to be accepted. How will we accomplish this? We're going to create a state machine.

State Machines

Put simply, a state machine looks for a series of inputs. When it reads an input, it changes state. Each step specifies the next state. When all of the required steps have been completed, in the correct order, it returns the desired result.

Consider purchasing a soda from a vending machine. First, you are required to insert the correct amount of money. If you only insert half, nothing happens. The correct input here is the full cost of the soda. Once you've inserted that, the next step is to choose which soda you want. The soda machine waits for this input before continuing. If you choose a cola, the machine will then proceed to dispense a cola. Once that is complete, the machine returns to it's original state: waiting for the correct amount of money to be inserted so it can repeat the steps once again.



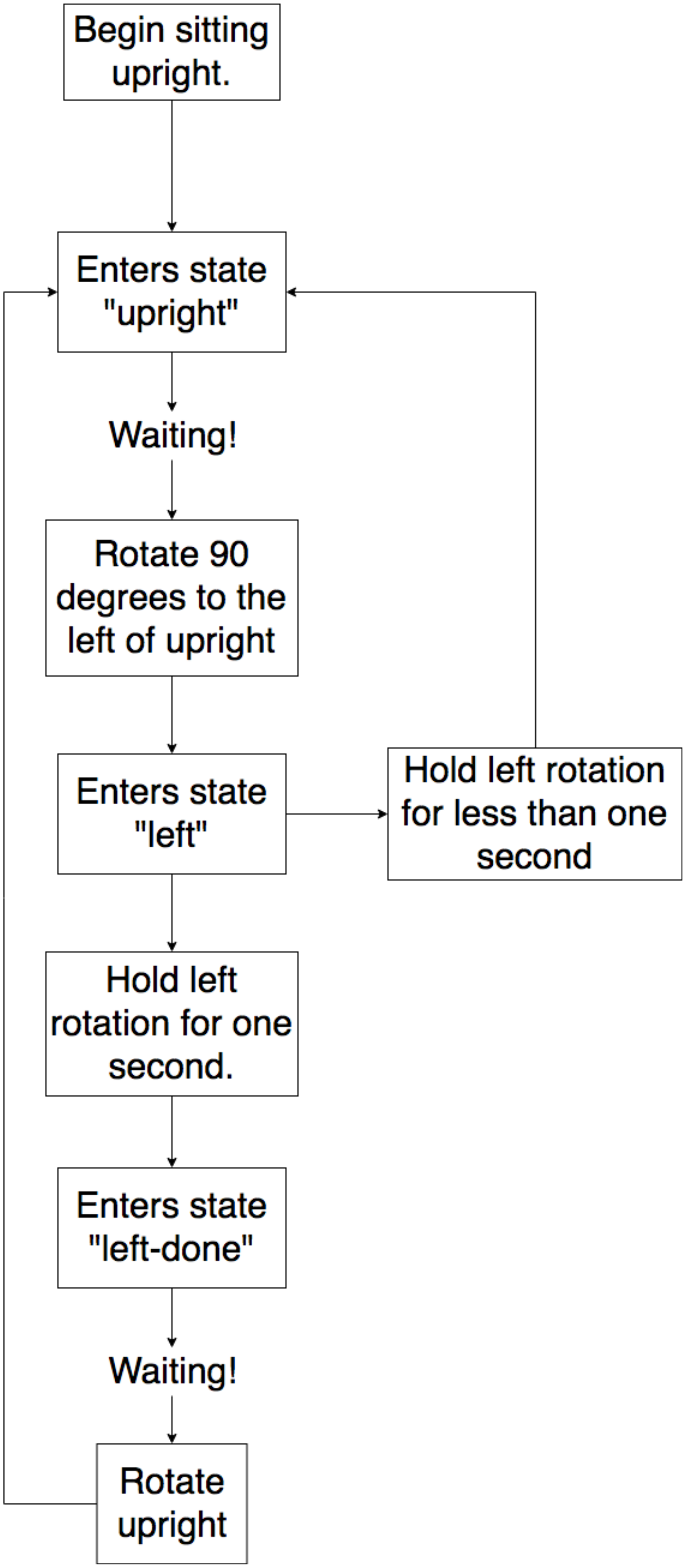
How does this apply to code? If you would like to require a series of inputs in a certain order, instead of a single input, you'll want to create a state machine. For example, if you wanted to use rotating your Circuit Playground Express as an input, you could set it so any time it is rotated to the left, it recognises that as an input and spams the result anytime it's in that position. However, if you wanted to have it see rotating left as an input once, and require it be rotated back before allowing the left rotation input to work again, you'd need a state machine. That's what we're going to do!

The State of Spoka

The first thing we need to do is decide what we want our state machine to look like. First, we want our code to first require a rotation of approximately 90 degrees to the left or right of the upright position. Next, it should require the lamp to be held in that rotated state for one second. Last, it should require the lamp to be placed in either the opposite rotation state or the upright state before the next rotation input in the same direction can be started.

Here is a list of the steps for using the left rotation as an input:

1. Begin upright or rotated to the right.
2. Rotate 90 degrees to the left of the upright position.
3. Hold for one second.
4. Rotate upright or to the right.


Begin sitting upright.
Enters state "upright"
Waiting!
Rotate 90 degrees to the left of upright
Enters state "left"
Hold left rotation for less than one second
Hold left rotation for one second.
Enters state "left-done"
Waiting!
Rotate upright

The right rotation will be the same steps, with right swapped for left.

We'll start with the left rotation input. Here is what our code looks like. Download the file and load it on your CPX

```
# SPDX-FileCopyrightText: 2018 Kattni Rembor for Adafruit Industries
#
# SPDX-License-Identifier: MIT

import time

from adafruit_circuitplayground.express import cpx


# pylint: disable=redefined-outer-name


def upright(x, y, z):
    x_up = abs(x) < accel_threshold
    y_up = abs(y) < accel_threshold
    z_up = abs(9.8 - z) < accel_threshold
    return x_up and y_up and z_up


def left_side(x, y, z):
    x_side = abs(9.8 - x) < accel_threshold
    y_side = abs(y) < accel_threshold
    z_side = abs(z) < accel_threshold

    return x_side and y_side and z_side


state = None
hold_end = None

accel_threshold = 2
hold_time = 1

while True:
    x, y, z = cpx.acceleration
    if left_side(x, y, z):
        if state is None or not state.startswith("left"):
            hold_end = time.monotonic() + hold_time
            state = "left"
            print("Entering state 'left'")
        elif (state == "left"
              and hold_end is not None
              and time.monotonic() >= hold_end):
            state = "left-done"
            print("Entering state 'left-done'")
    elif upright(x, y, z):
        if state != "upright":
            hold_end = None
            state = "upright"
            print("Entering state 'upright'")
```

Let's check this out! Connect to the REPL so you can view the `print` statements. Begin with the board flat, facing upright. You may see "Entering state 'upright'". Rotate the board 90 degrees to the left. You should see "Entering state 'left'". Hold for 1 second, or until you see "Entering state 'left-done'". Then rotate back to facing upright, and look for "Entering state 'upright'". Now you've used our state machine!

So let's take a look at the code.

Functions

First, we create the functions that define what "left" and "upright" mean.

```
def upright(x, y, z):
    return abs(x) &lt; accel_threshold and abs(y) &lt; accel_threshold and abs(9.8
- z) &lt; accel_threshold


def left_side(x, y, z):
    return abs(9.8 - x) &lt; accel_threshold and abs(y) &lt; accel_threshold and
abs(z) &lt; accel_threshold
```

To identify the orientation of the board, we're using the accelerometer. The accelerometer provides an `(x, y, z)` tuple, which is the acceleration value, in meters per second squared (m/s^2), currently applied to the x, y and z axes. We can use this to determine what direction the board is pointing. When the board is flat with the front facing upright, acceleration returns `(0, 0, 9.8)`. This is because there is 0 acceleration on the x and y axes, and 9.8m/s^2 (gravity!) on the z axis. So, the first function uses this information to tell the code that `upright` means when the board is flat and facing up. The same concept applies to `left_side`, with the values altered to match the values when the board is pointed to the left. As gravity is -9.8m/s^2, we take the absolute value of `(x, y, z)` using `abs()` to avoid dealing with negative numbers in our math.

Variables

Next, we create some variables for use later: `state` and `hold_end`. Then, we set `hold_time` to `1`, and `accel_threshold` to `2`.

```
state = None
hold_end = None

accel_threshold = 2
hold_time = 1
```

`hold_time` is the amount of time you must hold the board in the rotated state to continue to the next step in the state machine. `accel_threshold` is the number used to determine how close to exactly-rotated you must have the board to activate the input. If the `accel_threshold` is 0, then it must be rotated to exactly the right spot to activate it. Experimentation led us to use 2, which provides a range for the orientation that successfully activates the input. If you find you're having trouble finding the correct angle to hold the board, you can increase this number. `accel_threshold` must be between `0` and `9.8`. Be aware that the higher the number, the easier it is to activate the input, so you may mistakenly activate it if you set the number too high. At `9.8`, any movement registers as a rotation. It is not recommended to set it that high.

The Loop

Next we begin our loop. The first thing we do is call and assign acceleration. Next, we begin our state machine. At any given point in time, the code is looking to see whether the board is pointing to the `left_side` or `upright`. Within that, we begin to use that information to work through the steps of our input.

If we rotate to the left side, it checks to make sure that the `state` is either `None` (as we assigned on startup), or doesn't start with `"left"`.

```
if left_side(x, y, z):
        if state is None or not state.startswith("left"):
            hold_end = time.monotonic() + hold_time
            state = "left"
            print("Entering state 'left'")
```

If either one of these conditions is met, it assigns `hold_end = time.monotonic() + hold_time`, and the code enters the `"left"` state. This meaning `hold_end` is equal to the current time plus the `hold_time` and `state = "left"`.

If neither of the first two conditions is true, it checks for three other conditions.

```
        elif (state == "left"
              and hold_end is not None
              and time.monotonic() &gt;= hold_end):
            state = "left-done"
            print("Entering state 'left-done'")
```

Is the `state` equal to `"left"`, is `hold_end` not equal to `None`, and is the current time greater than or equal to `hold_end`. If all of these conditions is met, it begins to check whether or not the hold time has passed. Once it has, it enters the `"left-done"` state. At this point, we must return the lamp to the upright state to begin the left rotation part of our state machine again.

The last section checks to see if the board is upright.

```
    elif upright(x, y, z):
        if state != "upright":
            hold_end = None
```

```
            state = "upright"
            print("Entering state 'upright'")
```

If so, it checks to see if the state is not already upright before setting `hold_end = None` and `state = "upright"`. The reason for verifying that the state is not already upright is to avoid spamming the upright position, since the lamp spends most of its time upright.

And now our state machine can begin again!

CircuitPython Creature Friend

The Spoka lamp appears designed for Circuit Playground Express. The board fits perfectly into a groove in the bottom which holds it in place and the lamp is easy to hold in your hand. These things make it perfect for using motion to control it.

We're going to use three different inputs: double-tap, shake, and rotation. All three of these inputs are motion based and use the accelerometer. These inputs will control different modes, speeds, brightness and turning the lamp off.

We'll use:

- double-tap to change color modes
- rotate left to change brightness
- rotate right to change speeds
- and shake to turn the lamp off

The nine different modes that double-tap will cycle through are:

- a smooth rainbow cycle
- 7 different static solid colors
- and a cycle through the 7 solid colors (party mode!)

The speed changes we will code affect the speed of the cycle modes, and do not affect the solid colors.

As all three of these inputs are motion based and use the same sensor, under certain circumstances, one input can be mistaken for another. If this happens consistently, try performing one of the motions differently. For example, perhaps you are double-tapping the lamp while it is sitting on the table, but it is moving around enough that the shake input is triggering. In that case, try holding it in your hand and double-tapping it. The same goes for any input that is being triggered inadvertently. Identify which one it is and modify your motion to only trigger the input that you're actively trying to use.

What Worked and What Didn't

We planned ahead of time to use IR to control Sjopenna, and this proved to work perfectly. Our little creature friend Spoka, however, didn't have any specific plans to begin with, because we wanted to experiment with all the options to see what worked. So, the first thing we did was test different inputs.

The Circuit Playground Express fits snugly into the bottom of Spoka and mostly covers the capacitive touch pads. We tried adding a strip of copper tape to the side that would make contact with one of the pads, but the tape didn't stick to the surface. The lamp itself is not at all conductive so sensing touch through the lamp itself was out. We tried using the sound sensor to have it respond to loud noises, however, the CPX is sealed enough into the lamp that sound didn't reach it effectively. We tried using the light sensor as an input, but the amount of light needed to trigger it couldn't get through the lamp housing. In the end, we decided to use motion to interact with this lamp - tap, shake and rotation all use the accelerometer, and all three work really well!

The Code!

We've learned how to use `time.monotonic()` to create non-blocking code, how to create a state machine to use multi-step inputs, and how to use generators to allow for interruptible animation cycles. Now we'll put it all together.

Load the file on your Circuit Playground Express, and give it a try! Double-tap to switch between color modes. Rotate left and hold to change brightness. Rotate right and hold to change the speed of the rainbow modes. Shake to turn it off. Rotate left and hold while it's off to turn it back on.

```
# SPDX-FileCopyrightText: 2018 Kattni Rembor for Adafruit Industries
#
# SPDX-License-Identifier: MIT

import time
from rainbowio import colorwheel
from adafruit_circuitplayground.express import cpx


# pylint: disable=redefined-outer-name
def upright(x, y, z):
    return abs(x) < accel_threshold \
           and abs(y) < accel_threshold \
           and abs(9.8 - z) < accel_threshold


def right_side(x, y, z):
    return abs(-9.8 - x) < accel_threshold \
           and abs(y) < accel_threshold \
           and abs(z) < accel_threshold


def left_side(x, y, z):
    return abs(9.8 - x) < accel_threshold \
           and abs(y) < accel_threshold \
           and abs(z) < accel_threshold


# pylint: enable=redefined-outer-name


def cycle_sequence(seq):
    while True:
        for elem in seq:
            yield elem


def rainbow_lamp(seq):
    g = cycle_sequence(seq)
    while True:
        # pylint: disable=stop-iteration-return
        cpx.pixels.fill(colorwheel(next(g)))
        yield


def brightness_lamp():
    brightness_value = cycle_sequence([0.4, 0.6, 0.8, 1, 0.2])
    while True:
        # pylint: disable=stop-iteration-return
        cpx.pixels.brightness = next(brightness_value)
```

```
        yield


color_sequences = cycle_sequence([
    range(256),  # rainbow_cycle
    [0],  # red
    [10],  # orange
    [30],  # yellow
    [85],  # green
    [137],  # cyan
    [170],  # blue
    [213],  # purple
    [0, 10, 30, 85, 137, 170, 213],  # party mode
])

heart_rates = cycle_sequence([0, 0.5, 1.0])

brightness = brightness_lamp()

heart_rate = 0
last_heart_beat = time.monotonic()
next_heart_beat = last_heart_beat + heart_rate

rainbow = None
state = None
hold_end = None

cpx.detect_taps = 2
accel_threshold = 2
cpx.pixels.brightness = 0.2
hold_time = 1

while True:
    now = time.monotonic()
    x, y, z = cpx.acceleration

    if left_side(x, y, z):
        if state is None or not state.startswith("left"):
            hold_end = now + hold_time
            state = "left"
        elif (state == "left"
              and hold_end is not None
              and now >= hold_end):
            state = "left-done"
            next(brightness)
    elif right_side(x, y, z):
        if state is None or not state.startswith("right"):
            hold_end = now + hold_time
            state = "right"
        elif (state == "right"
              and hold_end is not None
              and now >= hold_end):
            state = "right-done"
            heart_rate = next(heart_rates)
            last_heart_beat = now
            next_heart_beat = last_heart_beat + heart_rate
    elif upright(x, y, z):
        if state != "upright":
            hold_end = None
            state = "upright"

    if cpx.tapped or rainbow is None:
        rainbow = rainbow_lamp(next(color_sequences))

    if now >= next_heart_beat:
        next(rainbow)
        last_heart_beat = now
        next_heart_beat = last_heart_beat + heart_rate
```

```
    if cpx.shake(shake_threshold=20):
        cpx.pixels.brightness = 0
```

We've combined everything we learned to create this amazingly interactive lamp! We've already learned in detail how to do everything we use in this program. Now we'll take a quick look at the code so we can see how it all fits together.

The Code!

We start with the `wheel` code, and our definitions of `upright`, `right_side` and `left_side`.

Next, we include all of our generators (). We have our special `cycle_sequence` generator and `rainbow_lamp`. We also have `brightness_lamp` which includes the different brightness levels. Then we have `color_sequences` and `heart_rates`.

We assign `brightness_lamp()` to a variable so we can use it later in the code.

The next section sets up the `time.monotonic()` variables ().

Following that, we create the `rainbow`, `state` and `hold_end` variables for later use.

Next, we set the code to look for double-taps and set the threshold for rotation orientation to `2`. We set the brightness on startup to 20% (expressed as `0.2`). We set the length of time required to hold in a rotated state to `1` second. If you'd like your state machine to require a different hold time, change this number!

With that, we start the loop! First, we get the current time and call acceleration.

Then we have our state machine (). If you rotate left and hold, it cycles to the next brightness level in the list. If you rotate right and hold, it uses some of our `time.monotonic()` variables to help with cycling to the next speed.

Next, the code waits for a double-tap to cycle to the next color mode.

The next section uses the current speed and our `time.monotonic()` variables to determine how fast to display the rainbow color modes, by determining how fast to call `next` on `rainbow`.

And the last section turns the lamp off if you shake it.

And that's it!

Now you have an interactive creature friend to light up your life in all kinds of colors!

CircuitPython Remote Lamp

We've designed the geometric Sjopenna's code to allow control from afar using a mini infrared remote control. Any remote will work if you properly decode the signals. This project is specifically coded for the Adafruit Mini Remote Control. We'll use the CircuitPython IR remote library to read and decode the IR signals.

This project uses many colors associated with different buttons on the IR remote. To keep the code as efficient as possible, we learned how to use a dictionary () to eliminate the need for a large block of `if` and `elif` statements. Now we're going to incorporate that into this code.

Regarding Rainbows

The way that the CircuitPython library for the infrared remote code is written, it is not possible to have a rainbow cycle setting. The library code is blocking which does not allow for the cycle generator that we used for Spoka to function properly. It continues only when the board receives an infrared signal, because the `read` function will wait indefinitely until it reads a signal. Due to the significant IR noise present in most environments, at first glance, the rainbow cycle appears to be functioning. As well, if you were to set a rainbow to a particular button, and then hold the button down constantly, it would spam the signal and cycle the rainbow. However, if you cover the IR sensor to block any signals, you'll find that the cycle does not progress. In the event that intermittent noise is received, the progression is jumpy and inconsistent. Given that we cannot predict the amount of IR noise present in any given environment, or expect you to hold down a button forever, we've chosen not to include a rainbow cycle setting in this code.

The Code!

Due to memory constraints we will not be using the same `Express` class that we used for Spoka - therefore we will import and initialise each library separately. However, using that class would only eliminate the need to import `neopixel` and `board`. The Adafruit IR Remote library, `adafruit_irremote`, and `pulseio` are not included in `Express` class and would have been required regardless.

Let's take a look at the code!

```
# SPDX-FileCopyrightText: 2018 Kattni Rembor for Adafruit Industries
#
# SPDX-License-Identifier: MIT

import adafruit_irremote
import board
import neopixel
import pulseio

pixels = neopixel.NeoPixel(board.NEOPIXEL, 10)

pulsein = pulseio.PulseIn(board.REMOTEIN, maxlen=120, idle_state=True)
decoder = adafruit_irremote.GenericDecode()

last_command = None

brightness_up = 95  # Up arrow
brightness_down = 79  # Down arrow

command_to_color = {  # button = color
    247: (255, 0, 0),  # 1 = red
    119: (255, 40, 0),  # 2 = orange
    183: (255, 150, 0),  # 3 = yellow
    215: (0, 255, 0),  # 4 = green
    87: (0, 255, 120),  # 5 = teal
    151: (0, 255, 255),  # 6 = cyan
    231: (0, 0, 255),  # 7 = blue
    103: (180, 0, 255),  # 8 = purple
    167: (255, 0, 20),  # 9 = magenta
    207: (255, 255, 255),  # 0 = white
    127: (0, 0, 0),  # Play/Pause = off
}

while True:
    pulses = decoder.read_pulses(pulsein, max_pulse=5000)
    command = None
    try:
        code = decoder.decode_bits(pulses)
        if len(code) > 3:
            command = code[2]
        print("Decoded:", command)
        print("-------------")
    except adafruit_irremote.IRNECRepeatException:  # Catches the repeat signal
        command = last_command
```

```
    except adafruit_irremote.IRDecodeException:  # Failed to decode
        pass

    if not command:
        continue
    last_command = command

    if command == brightness_up:
        pixels.brightness += 0.1
    elif command == brightness_down:
        pixels.brightness -= 0.1
    elif command in command_to_color:
        pixels.fill(command_to_color[command])
```

First we `import` the four libraries we'll be using in our code. Then we setup use of those libraries. Now we'll take a look at the next section.

Variables

First we assign `last_command` for use later. Then, we assign `brightness_up` to the IR command code associated with the up arrow on the IR remote, and `brightness_down` to the code for the down arrow.

Dictionary

Here is where we use the dictionary we learned about!

```
command_to_color = {        # button = color
    247: (255, 0, 0),       # 1 = red
    119: (255, 40, 0),      # 2 = orange
    183: (255, 150, 0),     # 3 = yellow
    215: (0, 255, 0),       # 4 = green
    87: (0, 255, 120),      # 5 = teal
    151: (0, 255, 255),     # 6 = cyan
    231: (0, 0, 255),       # 7 = blue
    103: (180, 0, 255),     # 8 = purple
    167: (255, 0, 20),      # 9 = magenta
    207: (255, 255, 255),   # 0 = white
    127: (0, 0, 0),         # Play/Pause = off
}
```

The `keys` are the IR codes for the eleven buttons we're using and the `values` are their associated `(r, g, b)` tuples. NeoPixel colors are represented using red, green and blue in values of 0 - 255 to determine the amount of a given color. For example, red is `(255, 0, 0)` as it does not contain any green or blue. When red, green and blue are all off, the values are `(0, 0, 0)`. We'll call this off, and use it to turn off the LEDs. We've used comments on each line to identify which button on the remote and assigned color the dictionary is referring to.

There are 21 buttons total on the Mini IR remote, and in this code, 13 of them are used. If you wanted to add more colors to this project, you can do so by extending the dictionary. You simply need to choose a button, add that `key` to the dictionary, and assign the desired `(r, g, b)` `value` to use them later in the code. Add a comment to the line to make it easier to remember what button and color you chose!

The Loop

The first two sections of code inside the loop are designed to read the incoming IR signals, decode them, and prepare them for practical use.

```
while True:
    pulses = decoder.read_pulses(pulsein, max_pulse=5000)
    command = None
    try:
        code = decoder.decode_bits(pulses)
        if len(code) &gt; 3:
            command = code[2]
        print("Decoded:", command)
        print("-------------")
    except adafruit_irremote.IRNECRepeatException:  # Catches the repeat signal
        command = last_command
    except adafruit_irremote.IRDecodeException:  # Failed to decode
        pass

    if not command:
        continue
    last_command = command
```

We must deal with the significant amount of IR noise, which shows up as single-value signals. The line `if len(code) > 3:` says the signal must be longer longer than three values before we bother to do anything with it. The decoded signal from each button on this remote is four numbers in a list format: `[0, 0, 0, 0]`. However, for use, you need only the third number from that list. So, when we get a code of the correct length, we assign `command` to be the third value from the list by assigning `command = code[2]`. (Remember, in CircuitPython, counting starts with `0`, so the third value is `2`!)

We've left in the two `print` statements so you can identify the command code for the unused buttons on the remote in the event you'd like to expand the project to use them.

```
        print("Decoded:", command)
        print("-------------")
```

Simply connect to the REPL, press a button, and use the resulting number in your code. If you choose to add to the color dictionary, this is how you can find the IR code to include as the `key`!

The last section is where we're telling the code what to do when a particular button is pressed.

When the lamp first lights up, the brightness is set to the maximum. Brightness is a percentage of 0 to 100 represented by a value of `0.0` - `1.0`, and is set using `pixels.brightness()`.

```
    if command == brightness_up:
        pixels.brightness += 0.1
    elif command == brightness_down:
        pixels.brightness -= 0.1
```

When we press the down arrow, assigned to `brightness_down`, it decreases the brightness by `0.1` each time. When we press the up arrow, assigned to `brightness_up`, it increases the brightness by `0.1`. This will not increase or decrease it beyond the minimum or maximum.

And finally, we get to call our dictionary!

```
    elif command in command_to_color:
        pixels.fill(command_to_color[command])
```

This last `elif` statement is checking to make sure that the `key` we're using is found in our dictionary. Without this check, your code will throw an error if you pressed a button not in use, as it cannot use a `key` or `value` that isn't found. The last line uses `pixels.fill()` and the values associated with the keys in the dictionary to turn our NeoPixels the chosen colors. This works because `pixels.fill()` expects the `(r, g, b)` values, we've associated the `(r, g, b)` `values` the `keys` in the dictionary. Like we learned earlier, this is how we take what would have been a huge block of `elif` statements and slimmed it down to one!

Now you can control a lamp from across the room with a little IR remote!