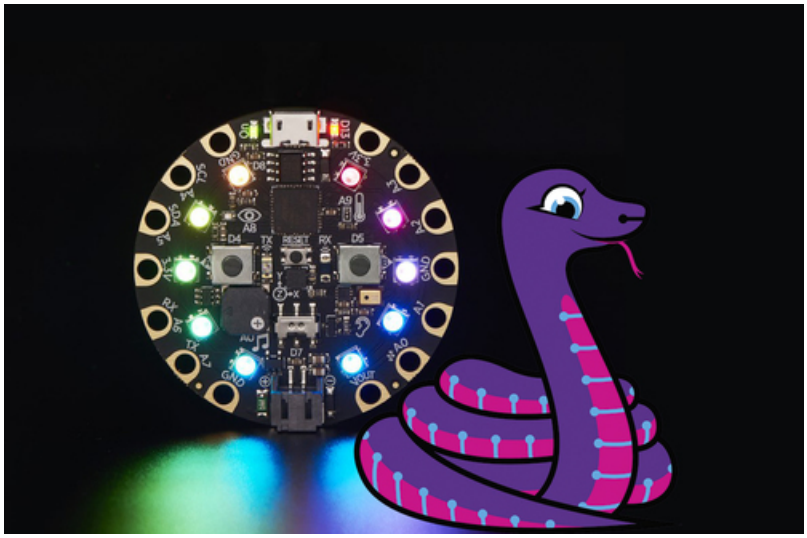
adafruit learning system

CircuitPython Made Easy on Circuit Playground Express and Bluefruit

Created by Kattni Rembor

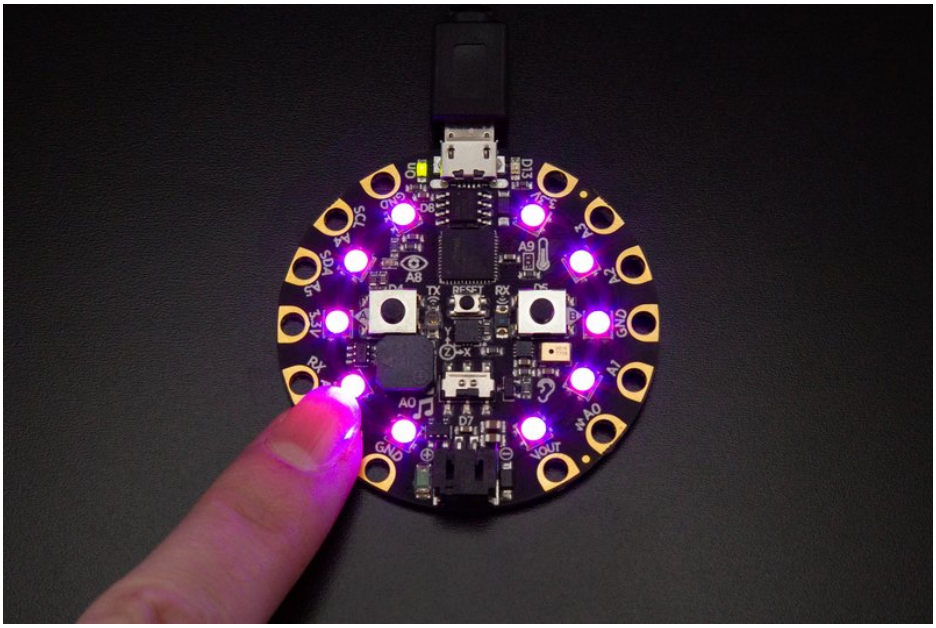


Last updated on 2020-05-05 11:48:22 AM EDT

Circuit Playground Library

The Circuit Playground Express (https://adafru.it/wpF), or CPX, and the Circuit Playground Bluefruit (https://adafru.it/Gpe), or CPB, have all kinds of sensors, buttons, switches and LEDs built into them. To top it off, they work with CircuitPython (https://adafru.it/zB0). Normally, using CircuitPython with a button or sensor requires setup in your code. Sometimes this means one line of code. Other times, it can mean several. Wouldn't it be nice to be able to skip all of that and get right to work? We've got you covered. Whether you're new to programming CircuitPython, or would like a simple way to include the Circuit Playground functionality in your code, the Circuit Playground CircuitPython library is exactly what you're looking for.

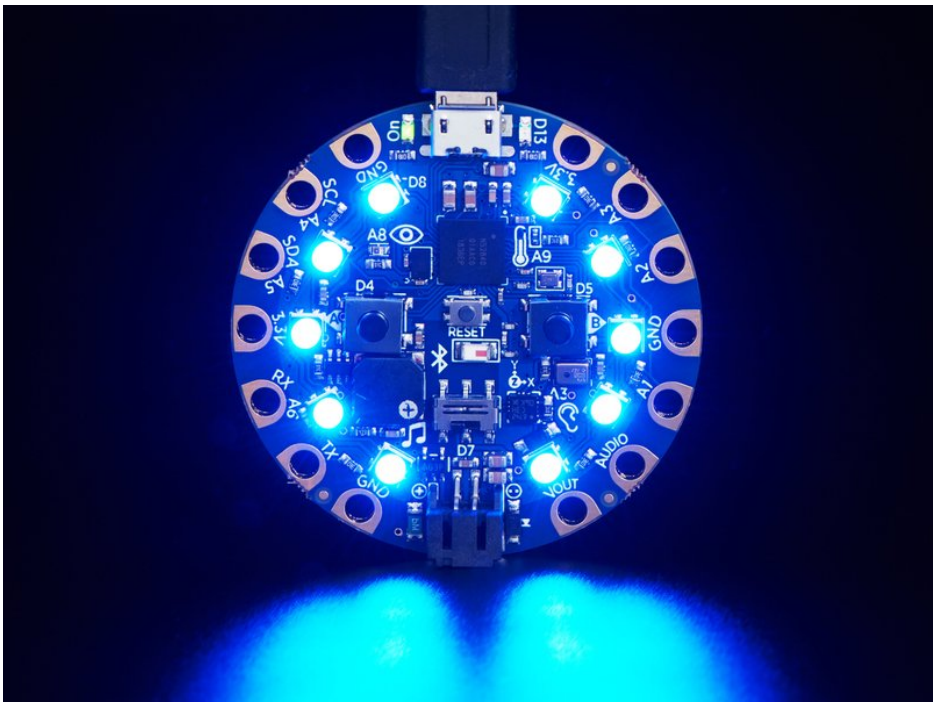
We've designed a CircuitPython library that makes it super easy to get started with Circuit Playground Express and Bluefruit. All of the setup normally required for much of what is built into the CPX or CPB is done for you. All you need to do is import this library, and you can immediately begin to tell the board what to do. This guide will go through each feature available in the library and provide examples of how to use it.



For the purposes of this guide, we'll refer to the Circuit Playground Express and Circuit Playground Bluefruit as "Circuit Playground", as the majority of the code within works on both boards with no changes needed. Where necessary, it will be made explicitly clear that a particular section works with a specific board.

Most of the images are of the Circuit Playground Express because most of the features of the Express and Bluefruit are in the same location. Images of both are included when a feature is in a different location on the Bluefruit.

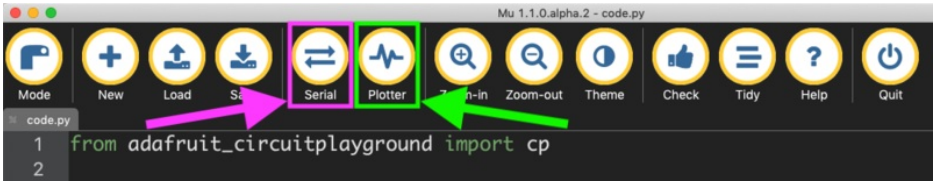
There's a few things you should do before going through this guide to make sure you're all set to go. Let's take a look!


D13
3.3V
A7
A2
GND
A1
AUDIO
VOUT
GND
TX
A6
RX
3.3V
SDA
A4
SCL
GND
D8
A8
A9
D4
D5
RESET
A3
D7

Before We Get Started

If you're new to programming and CircuitPython, check out the Welcome to CircuitPython guide (https://adafru.it/cpy-welcome).

We recommend using Mu as your code editor, as it has the serial console built right in, and you can get immediate feedback from your code right inside the editor. For help getting Mu setup, read through Installing Mu Editor (https://adafru.it/ANO). A few of the features of this library work really well with the plotter function available in the Mu editor. Be sure to install the latest version to make sure you have access to this feature of Mu.



Connecting to the serial console in Mu is as simple as clicking the serial button, shown above in magenta. To activate the plotter feature, click the Plotter button, shown above in green.

If you already have a favorite editor, feel free to use it for this guide. Many of the examples will utilise the serial console, so if you opt not to use Mu, and you've never connected to the serial console before, read through the Advanced Serial Console on Mac and Linux (https://adafru.it/AAI), or the Advanced Serial Console on Windows (https://adafru.it/AAH) for help getting connected.

Installing and Updating CircuitPython

This process is covered in the Installing CircuitPython section of the Welcome to CircuitPython guide (https://adafru.it/Amd). Even if your board arrived with CircuitPython installed, it may not be the latest version. You always want to have the most up-to-date version of CircuitPython on your board - this ensures the latest features and best functionality. Please take the time to go through the Welcome to CircuitPython: Installing CircuitPython (https://adafru.it/Amd) page (if you haven't already) and make sure you've got CircuitPython installed and up to date.

Updating CircuitPython is especially important on the Circuit Playground Express because the Circuit Playground Library is built into CircuitPython for the Express, and this guide expects the most up-to-date version of the library.

The Circuit Playground library and its dependencies are built into CircuitPython for the Circuit Playground Express. To use the library, no further action is needed.

Installing the Circuit Playground Library on Circuit Playground Bluefruit

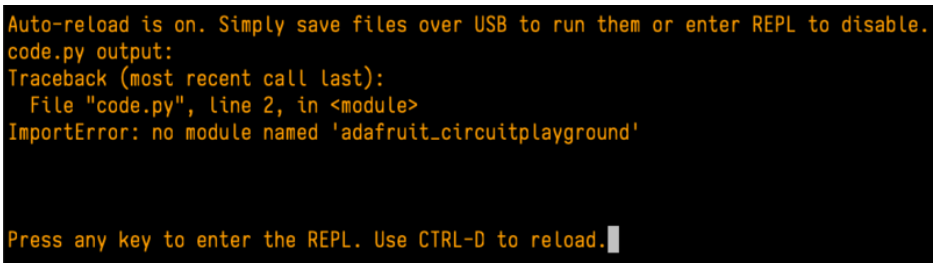
To use the Circuit Playground library with Circuit Playground Bluefruit, you must install the Circuit Playground library and its dependencies.

Before you can use the Circuit Playground library with the Circuit Playground Bluefruit, you must install the library and the modules it depends on. Follow the steps found on the Installing CircuitPython Libraries on Circuit Playground Bluefruit (https://adafru.it/l3c) section in the Circuit Playground Bluefruit guide to get all the necessary libraries installed.

The Circuit Playground library requires the following additional libraries

- **adafruit_bus_device**
- **adafruit_lis3dh**
- **adafruit_thermister**
- **neopixel**

If you try to run the code found within this guide without following these steps, the code will fail with the following error or one similar:



If you receive an `ImportError: no module named 'module_name` error, verify that you have installed all the libraries by going through the steps in the Installing CircuitPython Libraries on Circuit Playground Bluefruit (https://adafru.it/l3c) section again until your **lib** folder looks the same as the image found on that page.

Creating and Editing Code

This is covered in more detail in the Welcome to CircuitPython guide (https://adafru.it/BIN). However, since workflow is a key part of going through this guide, we're including a short explanation here.

Your Circuit Playground shows up on your computer as a USB drive called CIRCUITPY. You may already have some files on your CIRCUITPY drive. CircuitPython looks for specific files to run the code they contain, including **code.py**. We'll be putting each piece of code from this guide into code.py on your CIRCUITPY drive. This is easy to remember: **code.py is where your code lives**. As you progress through this guide, you have a couple of options to get the code from the guide onto your board.

1. You can **download the file**, **rename it to code.py** and **copy the file to your CIRCUITPY** drive, replacing the current code.py if one already exists.
2. You can **copy and paste the contents of the code from the guide into your current code.py** file on your **CIRCUITPY** drive using your editor. Be sure to **replace all the code currently in your code.py**. Do not add it to the end.

Both of these options work. It's entirely up to you which one to use. If you're unsure which to pick, give them both a try

and see which workflow is best for you!

Using the Circuit Playground Library

Regardless of which type of board you're using, to use the Circuit Playground library, simply include the following line at the beginning of code.py:

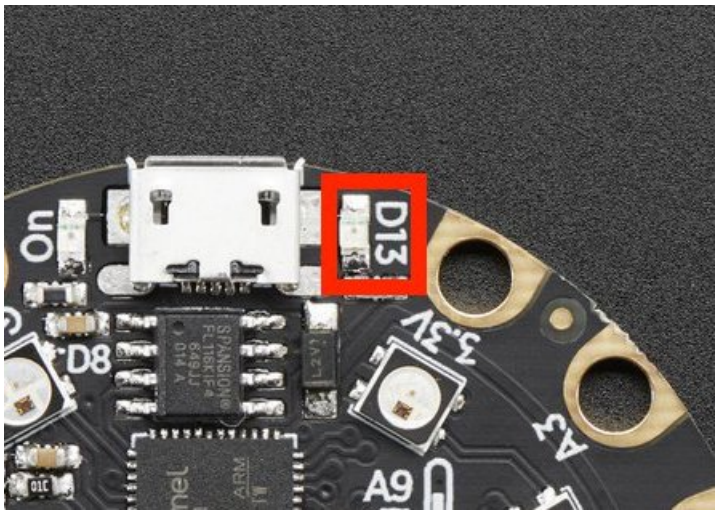
```
from adafruit_circuitplayground import cp
```

That's it! After that, you can begin telling the board what to do.

Now, we'll take a look at all of the different things you can do with this library. Let's get started!

Red LED

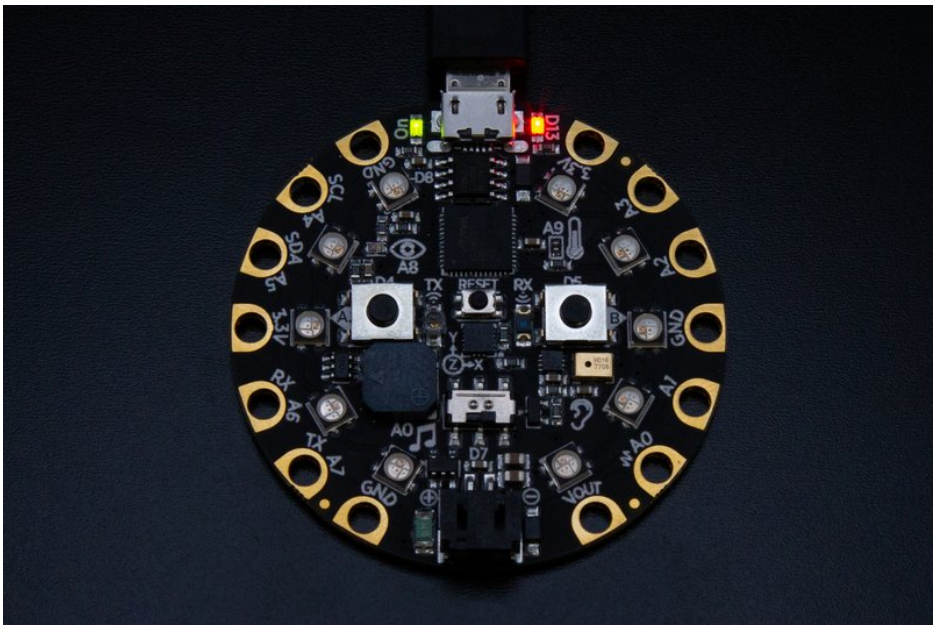
The Circuit Playground Express and Bluefruit have a little red LED next to the USB port. It's labeled D13. Though the images are of the Circuit Playground Express, the LED is in the same location on the Bluefruit. The first thing we're going to do is turn on that red LED.



First, we need to add the following code to **code.py**. Remember, if you need help with this, check here (https://adafru.it/C96).

```
"""This example turns on the little red LED."""
from adafruit_circuitplayground import cp

while True:
    cp.red_led = True
```

Red LED!



Now let's look at the code.

First we import the library with `from adafruit_circuitplayground import cp`.

Then we have a `while` statement. `while True:` essentially means, "Forever do:". `while True:` creates a loop. When there is a loop, the code will forever go through the code inside the loop. All code that is indented under `while True:` is "inside" the loop.

For the red LED, "on" and "off" are states referred to as `True` and `False` respectively. So, if you want to turn on the LED, you set it to `True`. If you want to turn it off, you set it to `False`. We want to turn on the LED. So let's set it to `True` by saying `cp.red_led = True`.

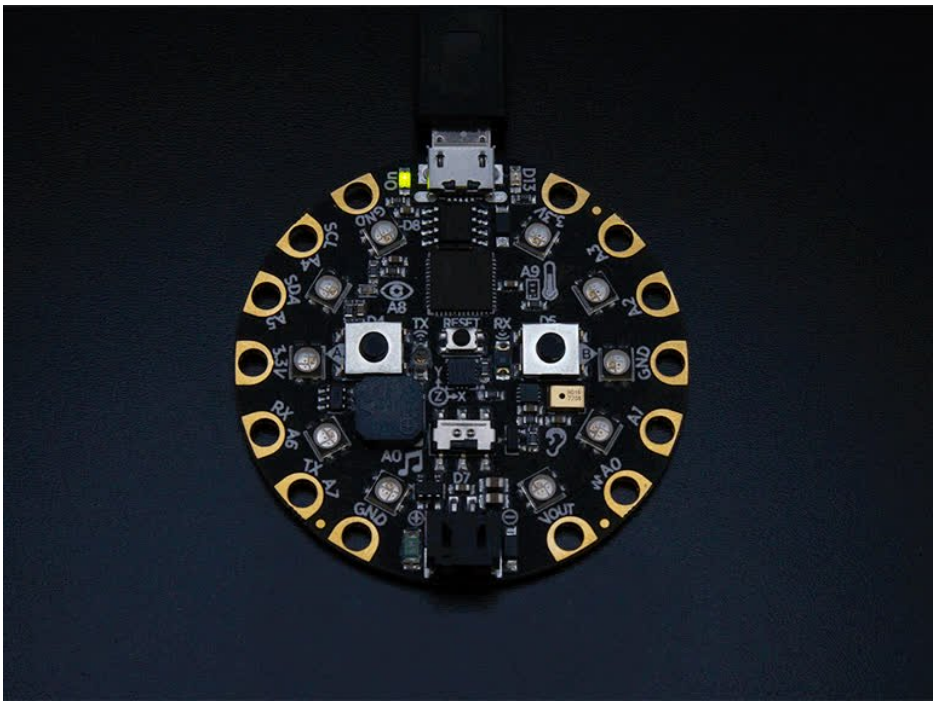
And that's it! You should be rewarded by the little red LED next to your USB connector turning on! But why stop there? Let's try something a little more fun.

Blinky!

In any programming language, the first piece of code any programmer writes is 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 called Blinky. Instead of simply writing code that prints out `hello`, we write code that blinks the LED! So, to welcome you to the world of programming, we're going to blink the little red LED. Let's take a look!

Add the following code to your **code.py**.

```
"""This is the "Hello, world!" of CircuitPython: Blinky! This example blinks the little red LED on
and off!"""
import time
from adafruit_circuitplayground import cp

while True:
    cp.red_led = True
    time.sleep(0.5)
    cp.red_led = False
    time.sleep(0.5)
```



It blinks!

In this program, we need another library as well: `time`. So, we import `time` and `cp`.

The first line inside our `while True:` loop is the same as the first line of our last program. We're turning on the red LED with `cp.red_led = True`. Next, we have `time.sleep(0.5)`. This tells the code to pause in the current state for 0.5 seconds. In other words, we're turning on the red LED and waiting with it on for 0.5 seconds. The next line, `cp.red_led = False`, turns the LED off. And the last line, `time.sleep(0.5)`, again tells the code to wait, this time with the LED off. Then it repeats forever - remember we're inside our `while` loop! And, when the LED turns on for 0.5 seconds and then off for 0.5 seconds, we have a blinking LED!

Try changing the numbers in the `time.sleep(0.5)` lines to change the speed of the blinking. You can slow down the blinking by replacing both `0.5`'s with a higher number, such as 1: `time.sleep(1)`. You can speed it up by replacing both `0.5`'s with a lower number, such as 0.1: `time.sleep(0.1)`. Or, try setting them to different times to give it a funky rhythm!

Red LED On = Red LED Off

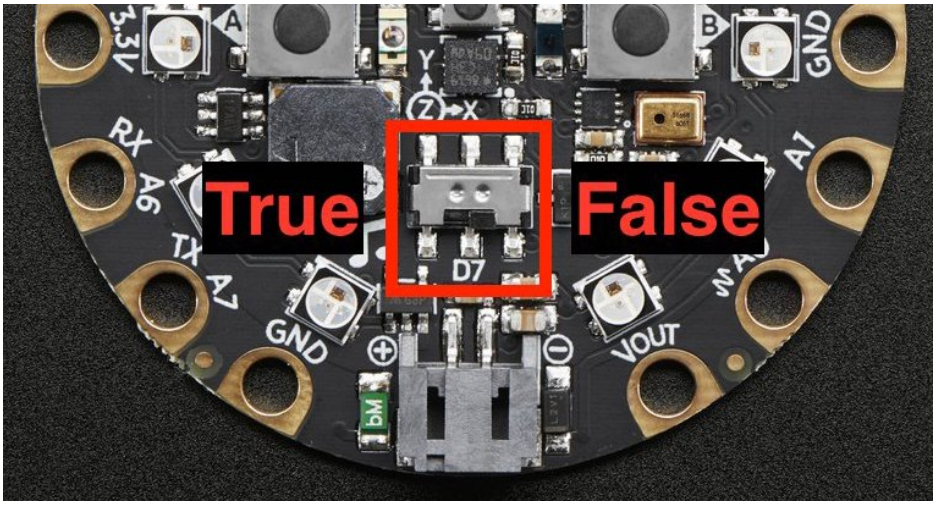
There's an even shorter way to do the same thing. Add the following code to your **code.py**.

```
"""This is the "Hello, world!" of CircuitPython: Blinky! This example blinks the little red LED on
and off! It's a shorter version of the other Blinky example."""
import time
from adafruit_circuitplayground import cp

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

This code simply tells the LED to cycle back and forth between on and off, or `True` and `False`, every 0.5 seconds. You can change the `time.sleep(0.5)` to a higher or lower number to slow down or speed up the blinking. That's it!

Slide Switch

The Circuit Playground Express and Bluefruit have a slide switch on it, above the battery connector. Though the images are of the Circuit Playground Express, the switch is in the same location on the Bluefruit. The slide switch doesn't control the power of the board. It is a switch that returns True or False depending on whether it's left or right. So, you can use it as a toggle switch in your code! Let's take a look.



Add the following code to your **code.py**. Remember, if you need help with this, check here (https://adafru.it/C96).

```
"""This example prints the status of the slide switch. Try moving the switch back and forth to see
what's printed to the serial console!"""
import time
from adafruit_circuitplayground import cp

while True:
    print("Slide switch:", cp.switch)
    time.sleep(0.1)
```

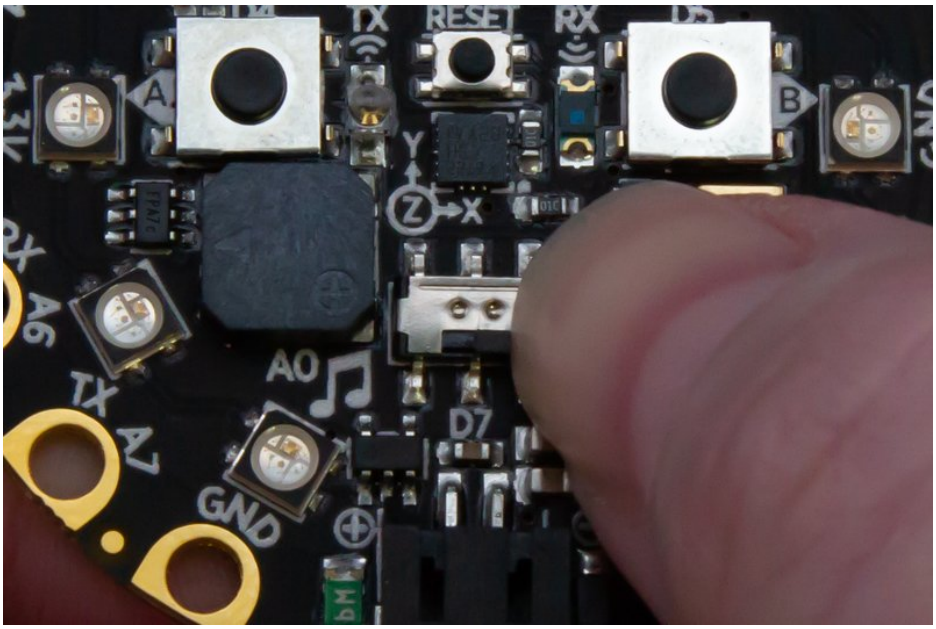
Open the serial console to see the switch status printed out. Try moving the slide switch back and forth to see the status change!

```
import time
from adafruit_circuitplayground import cp

while True:
    print("Slide switch:", cp.switch)
    time.sleep(0.1)
```

```
Slide switch: True
Slide switch: True
Slide switch: True
Slide switch: True
Slide switch: False
Slide switch: False
Slide switch: False
Slide switch: False
```

Let's take a look at the code. First we import `time` and `cp`.

Then, inside our `while` loop, we `print` the status of the switch to the serial console. This will print `True` if the switch is to the left, and `False` is the switch is to the right. We include a `time.sleep(0.1)` to slow down the printed output. To see the results, click the Serial button in Mu, or connect to the serial console if you're not using Mu. If the switch is to the left, you'll see `Slide switch: True` printing the serial console. If the switch is to the right, you'll see `Slide switch: False` printing to the serial console.



Simple enough, right? Now, let's do something with it!

Blinky Switch

We just learned how to turn the little red LED on and off. Now let's incorporate an input to control it. Since the switch

returns `True` or `False`, we can use it as an input.

Add the following code to your **code.py**.

```
"""This example uses the slide switch to control the little red LED."""
from adafruit_circuitplayground import cp

# This code is written to be readable versus being Pylint compliant.
# pylint: disable=simplifiable-if-statement

while True:
    if cp.switch:
        cp.red_led = True
    else:
        cp.red_led = False
```

After importing `cp`, our loop starts with an `if` statement. An `if` statement says, "if this event is happening, do the following." Our code says, if the switch is to the left, or `True`, turn on the red LED.

Note that we don't have to say `if cp.switch == True:`. The `True` is implied in the `if` statement.

This is followed by an `else` statement. And `else` statement says, "Otherwise, do the following." An `else` typically follows an `if`. Together they say, "If this is happening, do this first thing, otherwise, do the second thing." Our code says, when the switch is to the right, or `False`, turn off the red LED.

Now, try moving the switch back and forth. Your red LED will turn on and off!

True is True

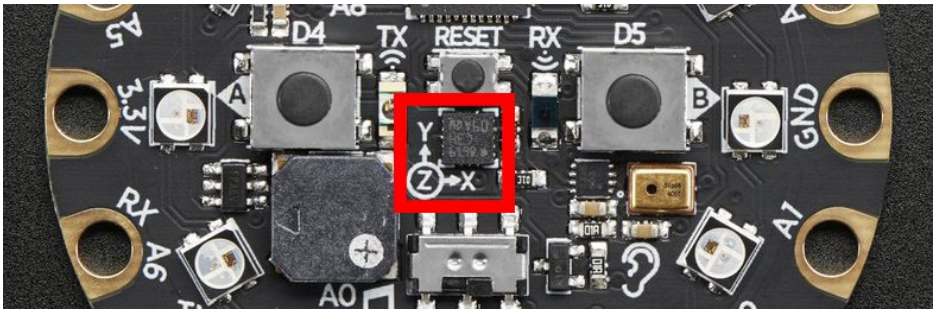
You may have noticed that when the switch is to the right, it's `True`, and when the LED is on, it is also `True`. We can use this to make our code even shorter. We started with the `if` / `else` block because it's easier to understand what's happening when it's written out. However, the following code does the same thing. Add the code to your **code.py**.

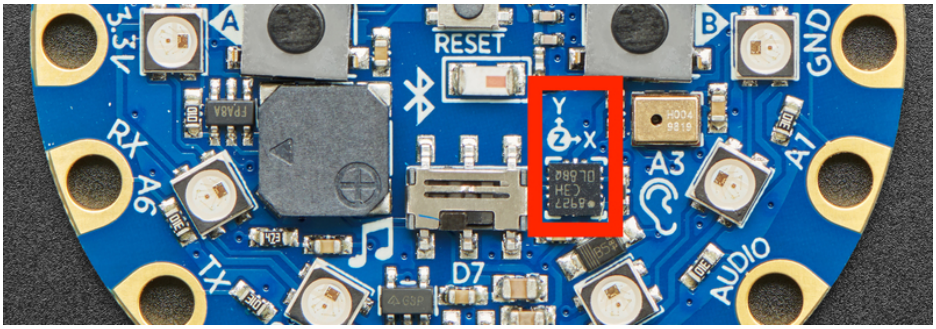
```
"""This example uses the slide switch to control the little red LED. When the switch is to the
right it returns False, and when it's to the left, it returns True."""
from adafruit_circuitplayground import cp

while True:
    cp.red_led = cp.switch
```

Whatever the switch is returning is what it will set the red LED to. So, if the switch is returning `True`, the LED is `True`. If the switch is `False`, the LED will be `False`. `True` is `True`, `False` is `False`. Move the switch back and forth and you'll still be turning the red LED on and off with this shorter code!

Tap

Circuit Playground Express and Bluefruit have an accelerometer built in which opens up all kinds of opportunities for inputs. One of those inputs is tap. You have the ability to tap your board to tell it to do something. There are two options: single tap and double tap. Single tap looks for one tap before reacting. Double tap looks for two taps before reacting. Let's take a look!

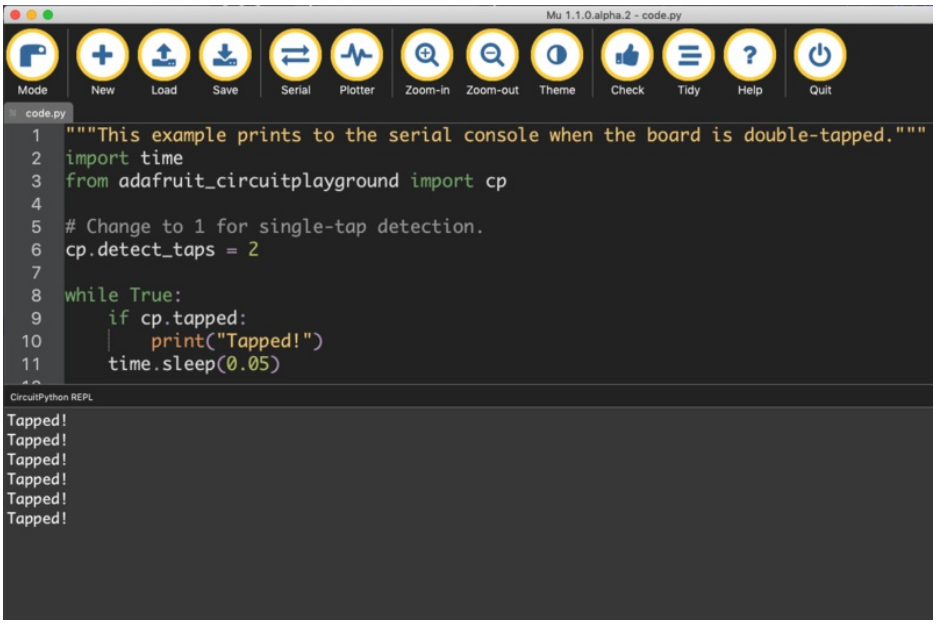




Add the following code to your **code.py**. Remember, if you need help with this, check here (https://adafru.it/C96).

```
"""This example prints to the serial console when the board is double-tapped."""
import time
from adafruit_circuitplayground import cp

# Change to 1 for single-tap detection.
cp.detect_taps = 2

while True:
    if cp.tapped:
        print("Tapped!")
    time.sleep(0.05)
```

Open the serial console to see when the board is double tapped. Try tapping the board twice to see it printed out!



First we import `time` and `cp`.

Then we set `cp.detect_taps = 2`. This tells the code to look for a double tap, or two taps, before responding.

Inside our loop, we have `if cp.tapped:`. The code tells `cp.tapped` that we're looking for 2 taps before responding. So, if the board is tapped twice, the response is to `print` `Tapped!` to the serial output. To see this, open the serial console, and tap your board twice. Tap twice. Tapped! We include a `time.sleep(0.05)` to prevent mistakenly detecting multiple double-taps at once.

Try changing `cp.detect_taps` to `1`. Tap the board once to see the same response!

Now, let's do something with it! Add the following code to your **code.py**:

```
"""This example turns on the little red LED and prints to the serial console when you double-tap
the Circuit Playground!"""
import time
from adafruit_circuitplayground import cp

# Change to 1 for detecting a single-tap!
cp.detect_taps = 2

while True:
    if cp.tapped:
        print("Tapped!")
        cp.red_led = True
        time.sleep(0.1)
    else:
        cp.red_led = False
```

Try tapping twice. Red LED!

Let's look at the code. First we import `time` and `cp`.

We'll keep `cp.detect_taps = 2` to tell the code to look for two taps.

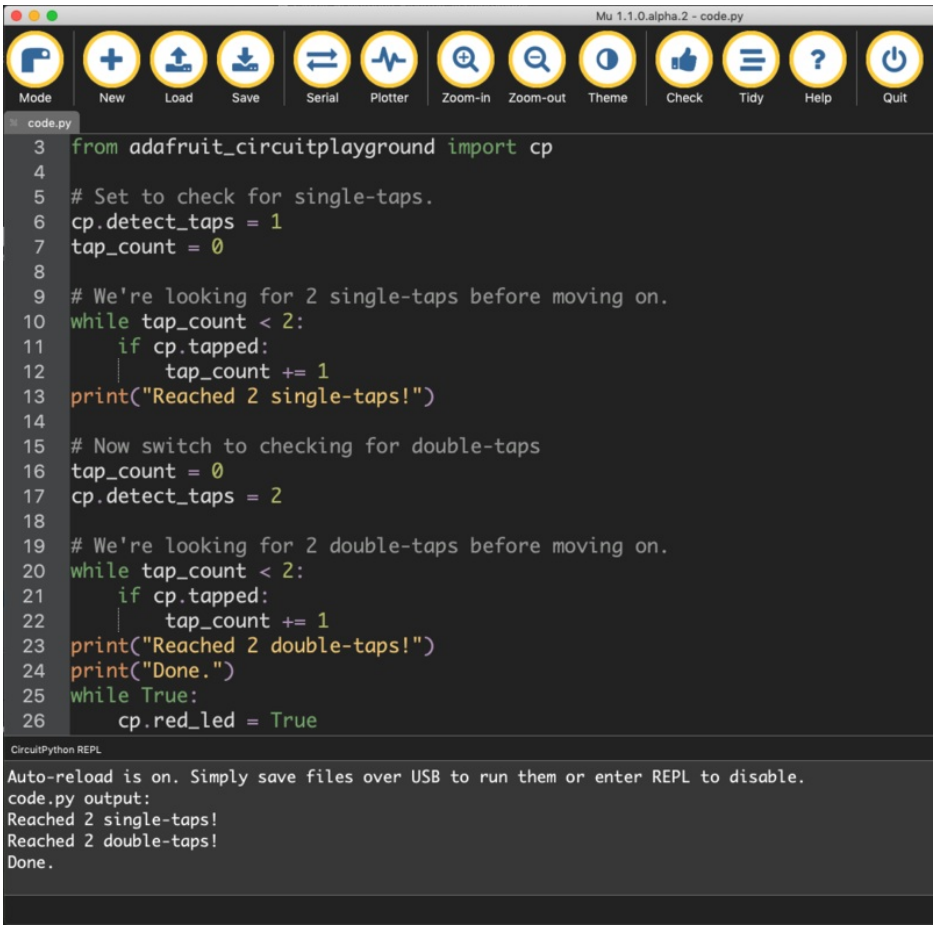
Inside our loop, we are checking to see `if` the board has been tapped twice. We still `print` to the serial output, so we can see if we've successfully tapped. But now, we've added turning on the red LED. Since the tap event is extremely quick, we've also included a `time.sleep(0.1)` so the red LED stays on long enough for us to see. Without it, it's a super quick flash. And we have our `else` to turn off the red LED when not tapping the board - otherwise it would turn on and never turn off.

Single Double

You can't detect a single tap and a double tap at the same time - it's a limitation of the hardware. You can include both single tap and double tap detection in one piece of code if you separate them with a delay of some sort. Let's take a look. Add the following code to your **code.py**.

```
"""This example shows how you can use single-tap and double-tap together with a delay between.
Single-tap the board twice and then double-tap the board twice to complete the program."""
from adafruit_circuitplayground import cp

# Set to check for single-taps.
cp.detect_taps = 1
tap_count = 0

# We're looking for 2 single-taps before moving on.
while tap_count < 2:
    if cp.tapped:
        tap_count += 1
print("Reached 2 single-taps!")

# Now switch to checking for double-taps
tap_count = 0
cp.detect_taps = 2

# We're looking for 2 double-taps before moving on.
while tap_count < 2:
    if cp.tapped:
        tap_count += 1
print("Reached 2 double-taps!")
print("Done.")
while True:
    cp.red_led = True
```

This code looks for two single-taps and then two double-taps to complete the sequence. So, if you single-tap the board twice, and then double-tap the board twice, you'll work though this code and see the messages printed out as you go!



Let's take a look at the code. First we import `cp` and then set it to look for single taps with `cp.detect_taps = 1`.

Then we create the variable `tap_count` and assign it to `0` with `tap_count = 0`. We're going to use this to keep track of how many times we've tapped the board. This is how we know when to move on to the next part of the code.

Our loop is different from our previous loops. This loop begins with `while tap_count < 2:`. It says, "keep looping through the following indented code until `tap_count` is greater than `2`." Since `tap_count` is currently `0`, we'll begin the loop. The code inside the loop says, "If the board has been tapped, increase `tap_count` by `1`." Each time you tap the board, it `prints` to the serial console, `Single-tap!` The first time you tap the board, `tap_count = 1`. The second time, `tap_count = 2`. `2` is not less than `2`, so after the second tap, the code stops working through this loop and moves on to the next section. The last thing we do before moving on is `print` to the serial console, `Reached 2 single-taps!` so we know we've reached the end of this section.

Next, we set `tap_count = 0` again since we're going to start looking for a new type of tap. Then we set the code to look for double taps with `cp.detect_taps = 2`.

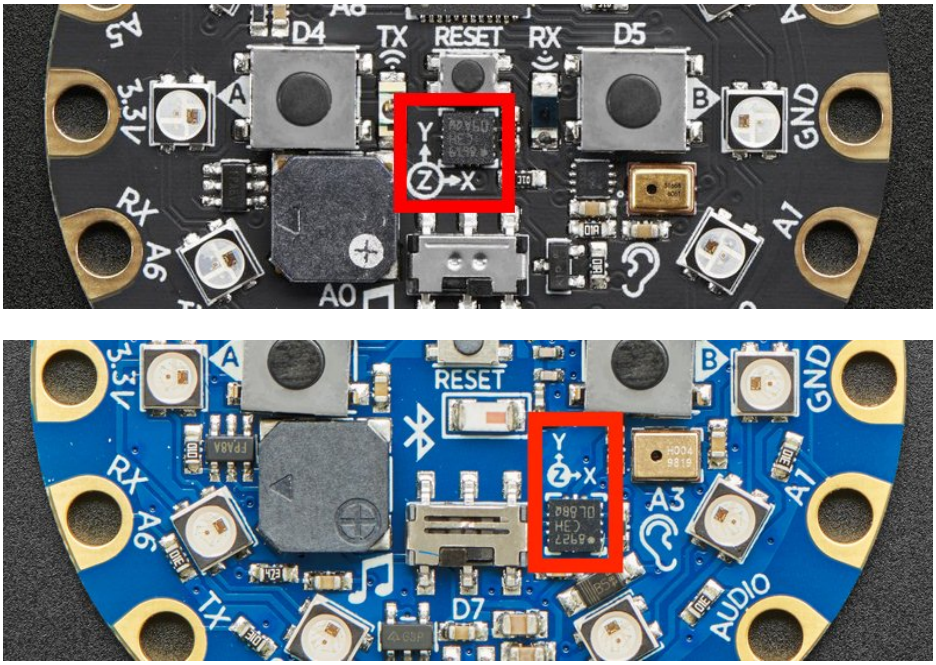
Our next loop is the same as the first. While `tap_count` is greater than `2`, check to see if the board is double tapped, and if it is, `print Double tapped!` and increase `tap_count` by `1`. Once it reaches `2`, the code moves on. Then we `print` to the serial console, `Reached 2 double-taps!`.

The last thing we do is print `Done`, and turn on the red LED so we know our code is finished.

This type of code could be used to create a Circuit Playground Express controlled combination lock where the combination is a series of taps. Get creative with it and see what you can come up with!

Shake

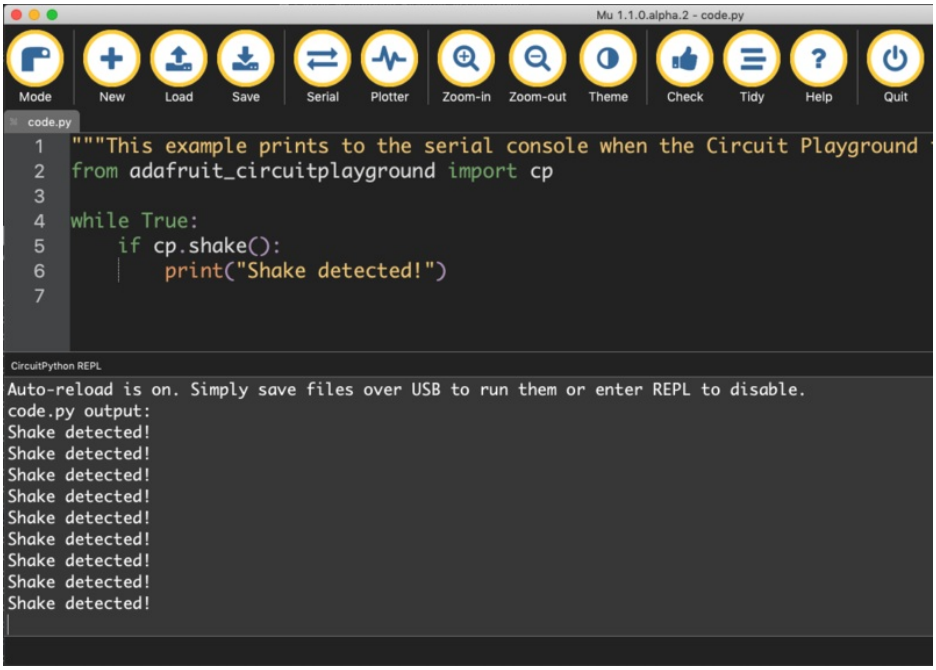
The Circuit Playground Express and Bluefruit accelerometer can detect other types of input actions besides taps. One of those inputs is shake. You have the ability to shake your board to tell it to do something. Let's give it a try!



Add the following code to your **code.py**. Remember, if you need help with this, check here (https://adafru.it/C96).

```
"""This example prints to the serial console when the Circuit Playground is shaken."""
from adafruit_circuitplayground import cp

while True:
    if cp.shake():
        print("Shake detected!")
```

Open the serial console and give the board a good shake. `Shake detected!`



Let's look at the code. First we import `cp`.

Inside our loop, we check to see if the board has been shaken with `if cp.shake():`. If the board is shaken, we `print` to the serial console, `Shake detected!`

Notice that there are parentheses after `cp.shake`. These are necessary for shake detection to work properly. Without them, your code will run, but it won't work properly. Make sure you include them!

Shake It Up A Little

Let's use shaking the board to turn on the red LED. Add the following code to your **code.py**.

```
"""This example flashes the little red LED when the Circuit Playground is shaken."""
from adafruit_circuitplayground import cp

while True:
    if cp.shake(shake_threshold=20):
        print("Shake detected!")
        cp.red_led = True
    else:
        cp.red_led = False
```

Shake the board. Red LED!

Let's look at the code. First we import `cp`.

Inside our loop, we check to see if the board has been shaken. However, we've added something to this line, `shake_threshold=20`. Sometimes you may find that the board doesn't respond to your shaking, or it responds too easily. You have the option to change the threshold to make it harder or easier to shake the board. The default threshold is 30. Decreasing the threshold makes it easier to have a shake detected. Increasing the threshold makes it harder to have a shake detected. The minimum value allowed is 10. 10 is the value when the board is not moving. So if you set the threshold to less than 10, the code will constantly return a shake detected even if the board is not moving.

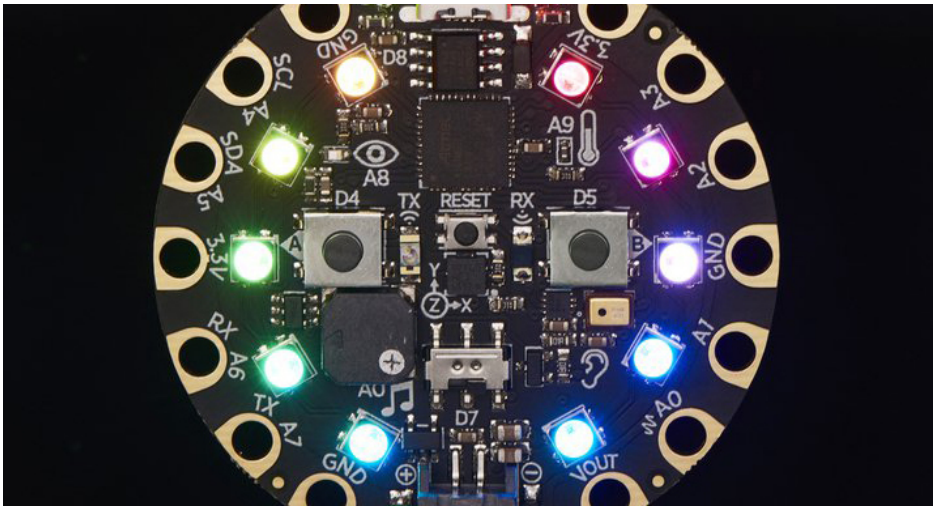
Set the threshold to any whole number above 10 to change the threshold to fit your needs.

In this case, we've included `if cp.shake(shake_threshold=20):` which lowers the threshold, making it easier to shake the board. If a shake over the threshold of 20 is detected, we `print` `Shake Detected!` and we turn on the red LED. Otherwise, we turn off the red LED with our `else` block.

Try changing the threshold to 40 and see what happens. Be aware, if you set the threshold too high, the shake will never be detected. Play around with it to find out what works best for you!

NeoPixels

The Circuit Playground Express and Bluefruit have ten RGB NeoPixel LEDs built in. Though the images are of the Circuit Playground Express, the LEDs are in the same location on the Bluefruit. They're located in a ring around the board, just inside the outer ring of alligator-clip-friendly pads. RGB means red, green and blue, and that means you can create any color of the rainbow with these LEDs!



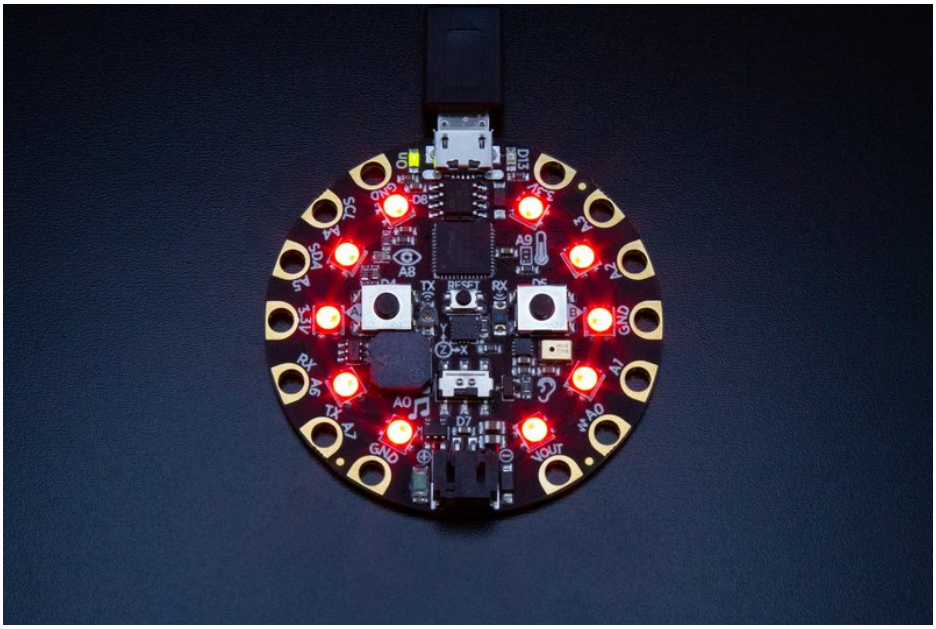
LED colors are set using a combination of **r**ed, **g**reen, and **b**lue, in the form of an (**R**, **G**, **B**) tuple. A tuple is typically a group of numbers. Each member of the RGB tuple is set to a number between 0 and 255 that determines the amount of each color present. Red, green and blue in different combinations can create all the colors in the rainbow! So, for example, to set the LED to red, the tuple would be (255, 0, 0), which has the maximum level of red, and no green or blue. Green would be (0, 255, 0), etc. For the colors between, you set a combination, such as cyan which is (0, 255, 255), with equal amounts of green and blue.

We won't make you wait any longer. Let's get started!

Add the following code to your **code.py**. Remember if you need help with this, check here (https://adafru.it/C96).

```
"""This example lights up all the NeoPixel LEDs red."""
from adafruit_circuitplayground import cp

while True:
    cp.pixels.fill((50, 0, 0))
```

Red lights!



First we import `cp`.

Inside our loop, we have `cp.pixels.fill((50, 0, 0))` which turns on all the pixels red at approximately 20% brightness. Remember, the maximum level of red is 255. That's really bright! So we've set it to a lower level of red so that it's not so blinding by setting it to `50`. The other two are 0, so there's no green or blue added into the mix yet. That's all there is to it!

Now, try changing the numbers to other values. For example, try `cp.pixels.fill((50, 50, 0))`. See what happens!

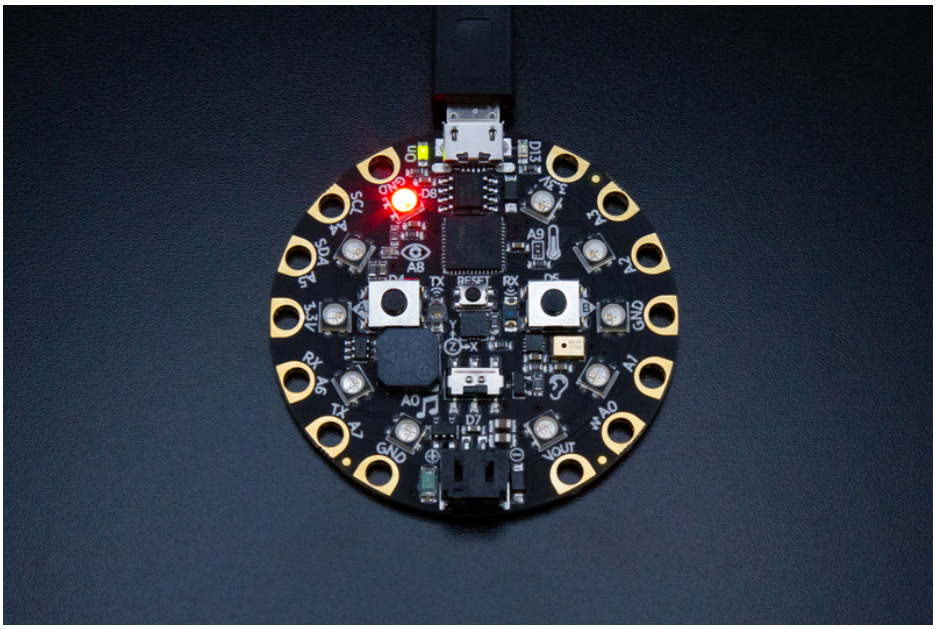
One Pixel, Two Pixel, Red Pixel, Blue Pixel!

We turned on all the pixels to the same color. But what if you want to control each one individually? We can do that!

Add the following code to your **code.py**:

```
"""This example lights up the first NeoPixel red."""
from adafruit_circuitplayground import cp

cp.pixels.brightness = 0.3

while True:
    cp.pixels[0] = (255, 0, 0)
```

Now only the first pixel is red!



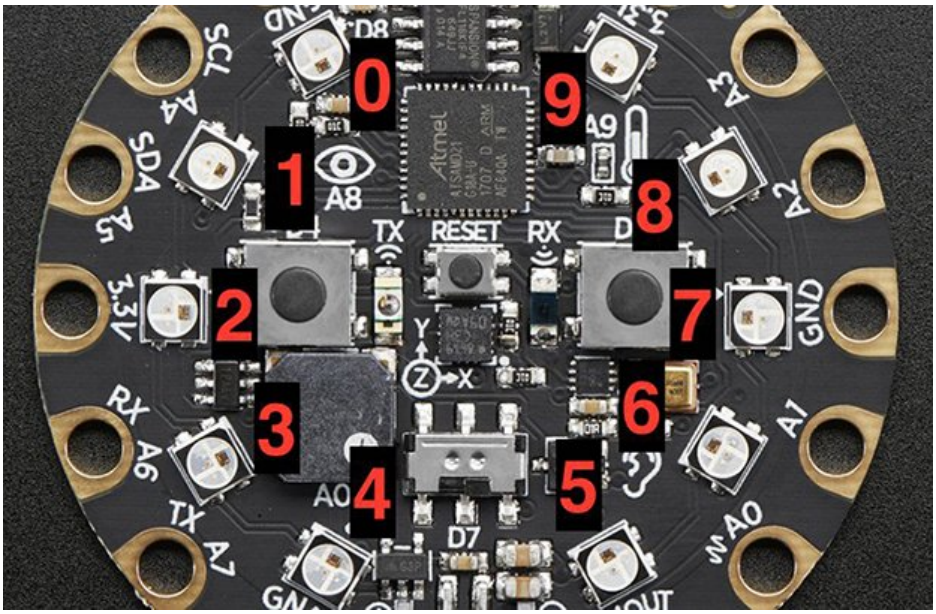
Let's look at the code.

First we import `cp`.

Next, we have a new line: `cp.pixels.brightness = 0.3`. Remember, we controlled brightness by using a lower number in the color tuple in the first piece of code. It's also possible to control brightness separately using `cp.pixels.brightness`. The brightness is set by a number between 0 and 1 that represents a percentage. So, when we set it to `0.3`, we are setting it to 30% brightness.

Inside our loop, we have `cp.pixels[0] = (255, 0, 0)`. Since we've set the brightness separately from the color, we are able to set the color to maximum red, or 255.

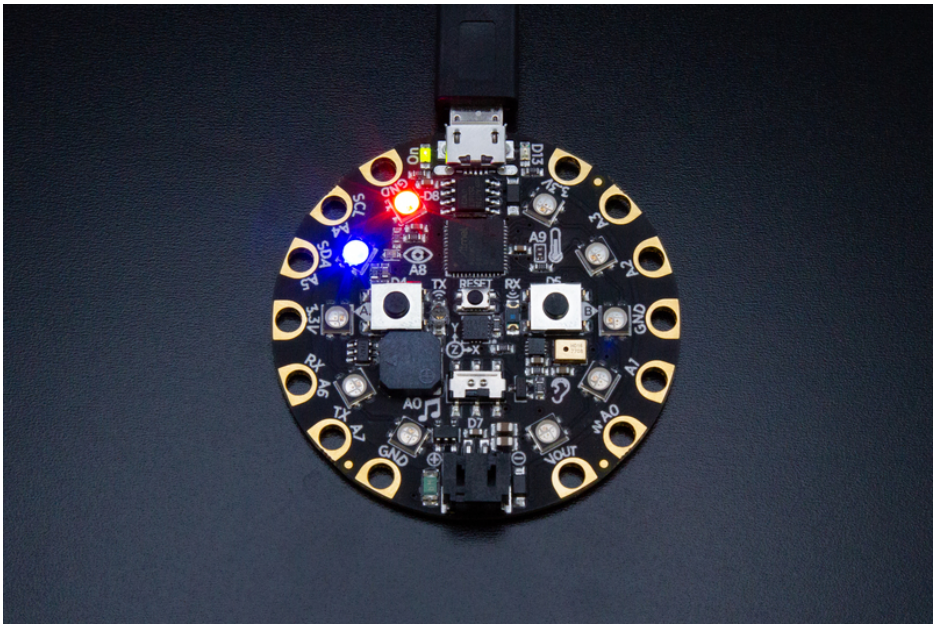
Notice we've set pixel number `0`, but it's turned on the first pixel. This is because CircuitPython begins counting with 0. So the first of something numbered in CircuitPython will always be 0.



Let's try setting the second pixel to blue. Remember, the second pixel will be pixel number 1. Add the following to your **code.py**.

```
"""This example lights up the first and second NeoPixel, red and blue respectively."""
from adafruit_circuitplayground import cp

cp.pixels.brightness = 0.3

while True:
    cp.pixels[0] = (255, 0, 0)
    cp.pixels[1] = (0, 0, 255)
```

Now your second pixel is blue.

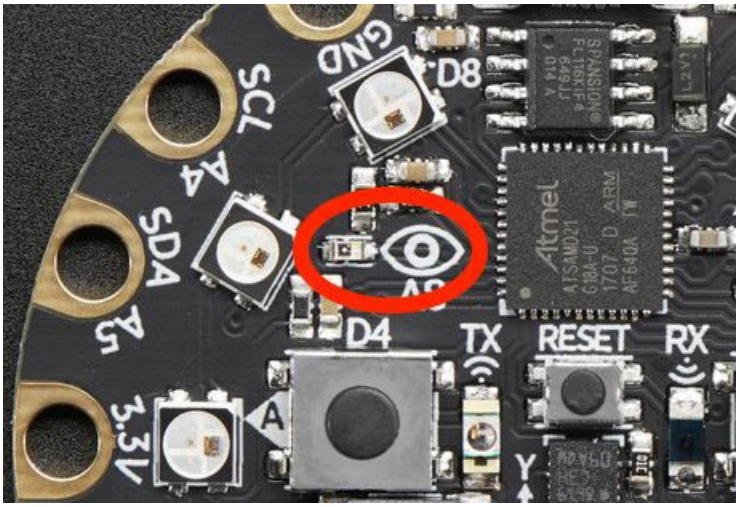


That's all there is to it! You can keep adding more pixels up through 9 to set all of them different colors.

Give it a try!

Light

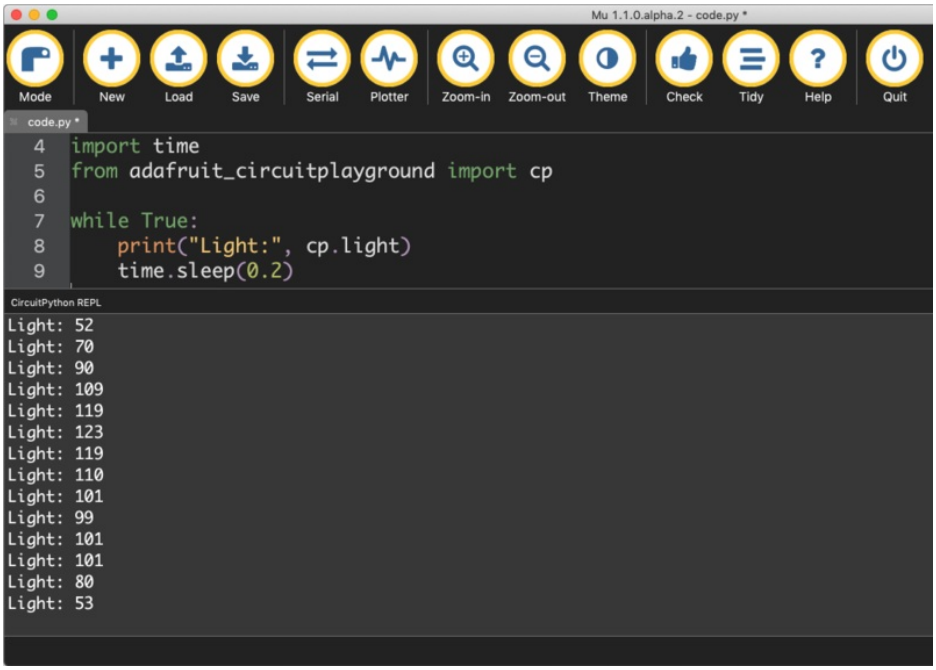
The Circuit Playground Express and Bluefruit have a light sensor on the right side, near the eye printed on the board. Though the images are of the Circuit Playground Express, the sensor is in essentially the same location on the Bluefruit. It senses the amount of ambient light and returns the light level based on that data. We've made it super easy to use. Let's take a look!



Add the following code to your **code.py**. Remember, if you need help with this, check here (https://adafru.it/C96).

```
"""This example uses the light sensor on your Circuit Playground, located next to the picture of
the eye. Try shining a flashlight on your Circuit Playground, or covering the light sensor with
your finger to see the values increase and decrease."""
import time
from adafruit_circuitplayground import cp

while True:
    print("Light:", cp.light)
    time.sleep(0.2)
```

Open the serial console and try shining a flashlight at your Circuit Playground. The printed values go up! If you place your hand over the board to block the light, the values go down.



Let's look at the code. First we import `time` and `cp`.

Inside our loop, we `print` to the serial console, `Light:` followed by the light value, `cp.light`. Then we have `time.sleep(1)` to slow down the speed at which it prints to the serial console. If it's too fast, it's really hard to read!

Plotting Light

Let's take a look at these values on the Mu plotter! Add the following code to your **code.py**:

```
"""If you're using Mu, this example will plot the light levels from the light sensor (located next
to the eye) on your Circuit Playground. Try shining a flashlight on your Circuit Playground, or
covering the light sensor to see the plot increase and decrease."""
import time
from adafruit_circuitplayground import cp

while True:
    print("Light:", cp.light)
    print((cp.light,))
    time.sleep(0.1)
```

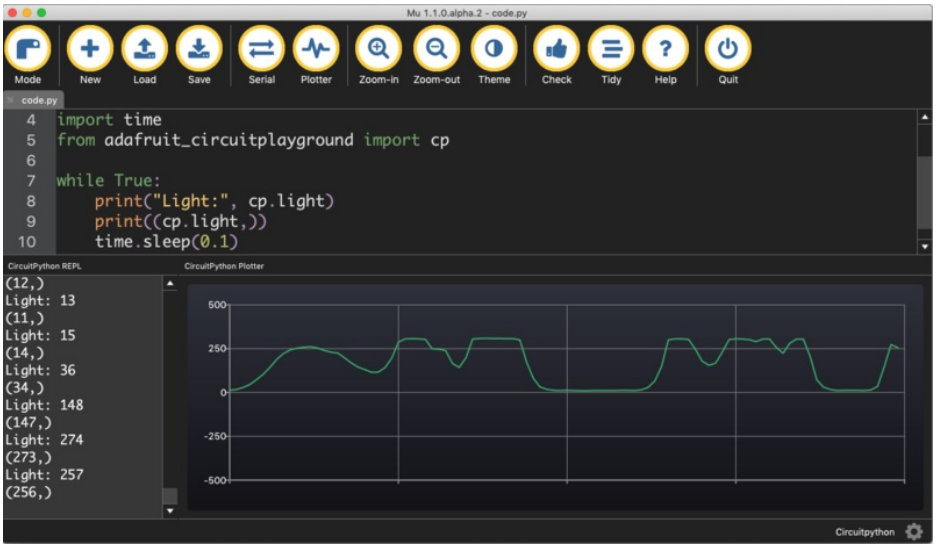
The code is almost identical, but we've added one line, `print((cp.light,))`.

Note that the Mu plotter looks for **tuple** values to plot. Tuples in Python come in parentheses `()` with comma separators. If you have two values, a tuple would look like `(1.0, 3.14)`. Since we have only one value, we need to have it print out like `(1.0,)` - note the parentheses *around* the number, and the *comma* after the number. Thus the extra parentheses and comma in `print((cp.light,))`.

As well, the Mu plotter requires that the tuple value be on a line all its own. That's why we can't simply add extra parenthesis and a comma to the `print("Light:", cp.light)` line. The plotter doesn't know what to do with it if there's other information in there.

Click on the Plotter button on the top of Mu to see the plotter. Try shining a flashlight on your Circuit Playground and

watch the plotter line go up! Remove or block the light with your hand to see it go down. Have fun with it!

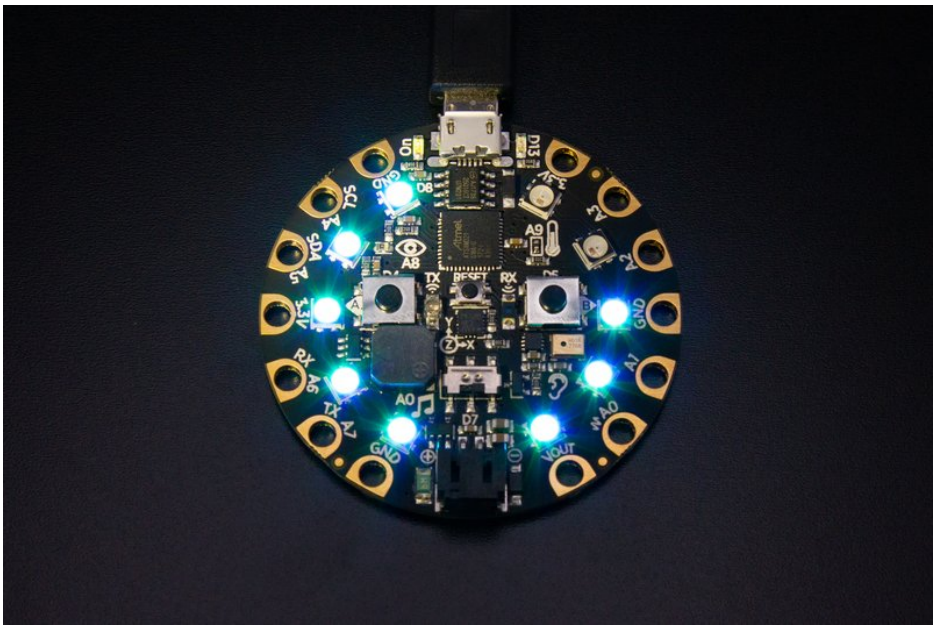


NeoPixel Light Meter

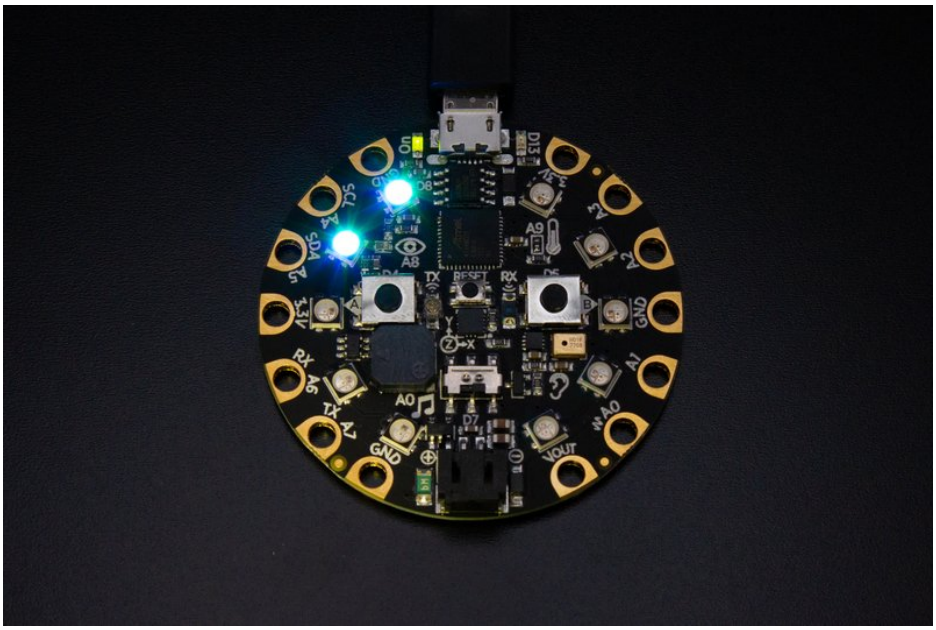
You can also use the light values to create a fun light meter using the NeoPixels on your Circuit Playground! Add the following code to your **code.py**:

```
"""
This example uses the light sensor on the Circuit Playground, located next to the picture of the
eye on the board. Once you have the library loaded, try shining a flashlight on your Circuit
Playground to watch the number of NeoPixels lit up increase, or try covering up the light sensor
to watch the number decrease.
"""

import time
from adafruit_circuitplayground import cp

cp.pixels.auto_write = False
cp.pixels.brightness = 0.3


def scale_range(value):
    """Scale a value from 0-320 (light range) to 0-9 (NeoPixel range).
    Allows remapping light value to pixel position."""
    return round(value / 320 * 9)


while True:
    peak = scale_range(cp.light)
    print(cp.light)
    print(int(peak))

    for i in range(10):
        if i <= peak:
            cp.pixels[i] = (0, 255, 255)
        else:
            cp.pixels[i] = (0, 0, 0)
    cp.pixels.show()
    time.sleep(0.05)
```

Now try shining the flashlight on your Circuit Playground and watch the LEDs light up!



Slowly remove the light to watch the number of LEDs lit up slowly go down.



Let's take a look at the code. First we import `time`, and `cp`.

Next, we set `cp.pixels.auto_write = False`. This means that anything we tell the LEDs to do will not happen automatically. By default, this is set to `True`. This means, we tell the LEDs to turn on, and they turn on. If it's set to `False`, it means we have to include `cp.pixels.show()` after anything we try to tell the LEDs to do. This is required for this code to work since the LEDs turn on based on the light values.

We set the `brightness` to `0.3`, or 30%.

Next we have a helper function called `scale_range`. The light values are approximately 0-320 but there are only 10 NeoPixels. So, we include a helper function that scales the 0-320 range to 0-9 so we can map light levels to pixel position.

Our loop begins with setting `peak = scale_range(cp.light)`. Then we print the `cp.light` values and the `peak` values.

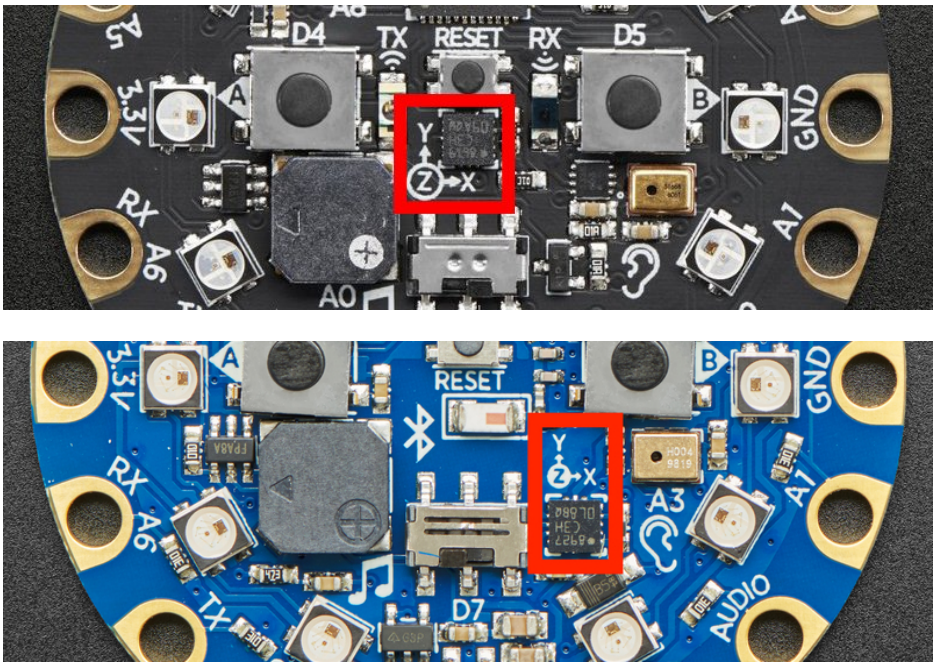
The next section takes the `peak` value and says for the total number of LEDs, whatever number `peak` is equal to or less than, light up that many LEDs, and otherwise turn them off. So, if peak is 4, light up 4 LEDs!

Then we have `cp.pixels.show()` to make the LEDs light up. And a `time.sleep(0.05)` to create a little delay.

You can change the number values in `cp.pixels[i] = (0, 255, 255)` to change the color of the light meter. Give it a try!

Acceleration

The Circuit Playground Express and Bluefruit both come with an accelerometer near the center of the board. This sensor can provide acceleration values for the x, y and z axes in addition to taps and shakes. The values returned are in m/s^2 (meters per second-squared). An axis is an invisible line going through the center of the accelerometer in the center of your board. The x axis is across the board, left to right. The y axis is across the board, top to bottom. The z axis is straight through the board front to back. The values can be grouped together in a Python tuple: `(x, y, z)`.



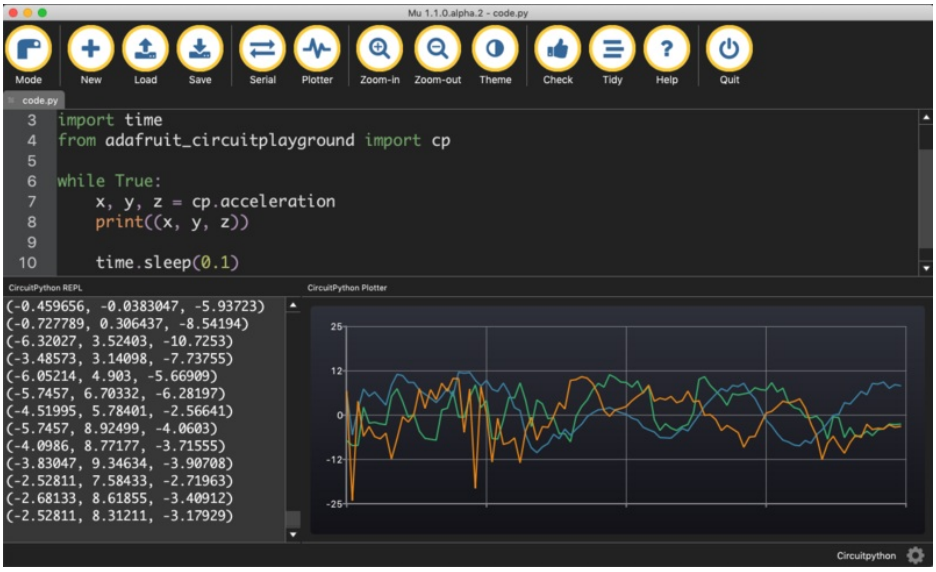
An accelerometer measures acceleration. You can read more about acceleration here (https://adafru.it/BnC). When the board is held still in any given position, it is still being affected by gravity. Gravity is $-9.8m/s^2$. So, at any given point in time, that value is being applied downward. For example, the values returned if the board is laying flat, facing up, are (0, 0, 9.8), because gravity is pulling on the sensor along the z axis. If you were to pick up the board and shake it, you'll find that you get much higher values. This is because the force with which you are shaking the board causes increased acceleration to be applied to the sensor along whichever axes you are shaking it.

Add the following code to your **code.py**. Remember, if you need help with this, check here (https://adafru.it/C96).

```
"""
This example uses the accelerometer on the Circuit Playground. It prints the values. Try moving
the board to see the values change. If you're using Mu, open the plotter to see the values plotted.
"""
import time
from adafruit_circuitplayground import cp

while True:
    x, y, z = cp.acceleration
    print((x, y, z))

    time.sleep(0.1)
```

Open the serial console to see the x, y and z values printed out. Try moving the board around to see the values

change!



Let's take a look at the code. First, we `import` `time` and `cp`.

Inside our loop, we assign `x, y, z = cp.acceleration`. Since acceleration values are a 3-member tuple (x, y, z), you need to assign three variables to `cp.acceleration` to get those three values. We've chosen `x`, `y` and `z` because those are the three axes represented by `cp.acceleration`.

Then we `print((x, y, z))`. We include a `time.sleep(0.1)` to slow down the printed values - if they `print` too quickly it's difficult to read.

Since `(x, y, z)` is already a tuple, and we aren't printing any labels for the values, we can use the Mu plotter with the code without any changes. Click on the Plotter button on the top of Mu to see the plotter. Try moving the board around to watch the plotter lines change!

Color Glow Accelerometer

You can use the acceleration values to make a fun light up project with the NeoPixels. There are three acceleration values, and the LEDs have three color values. Let's see what we can do with that!

Add the following code to your **code.py**.

```
"""If the switch is to the right, it will appear that nothing is happening. Move the switch to the
left to see the NeoPixels light up in colors related to the accelerometer! The Circuit Playground
has an accelerometer in the center that returns (x, y, z) acceleration values. This program uses
those values to light up the NeoPixels based on those acceleration values."""
from adafruit_circuitplayground import cp

# Main loop gets x, y and z axis acceleration, prints the values, and turns on
# red, green and blue, at levels related to the x, y and z values.
while True:
    if not cp.switch:
        # If the switch is to the right, it returns False!
        print("Slide switch off!")
        cp.pixels.fill((0, 0, 0))
        continue
    R = 0
    G = 0
    B = 0
    x, y, z = cp.acceleration
    print((x, y, z))
    cp.pixels.fill(((R + abs(int(x))), (G + abs(int(y))), (B + abs(int(z)))))
```

Move the slide switch to the right if it isn't already. Lights! Now move the board in different directions to see the colors change!

Let's take a look at the code. First we import `cp`.

Inside our loop, we start by checking to see `if` the switch is to the left. If it is, we `print` `Slide switch off!` and turn off all the LEDs. This creates an "off switch" for the project in case you'd like to leave it sitting around but not have the lights on. `continue` tells the code to keep checking the switch until the state changes, i.e. you move the switch to the right. Once that happens, we move on to the rest of the code.

Next we have the `else` block. First, we create three variables, `R`, `G` and `B`. We're going to use these to set the colors. We assign them to `0` to start. Then, we assign `x, y, z = cp.acceleration` and print the values. If you look at the serial output, you'll see how fast the values are scrolling. This is why we typically include a `time.sleep()` in the code, to slow those values down to a readable speed. However, this project works best without a `sleep`, so we've left it out.

The last line fills the LEDs with RGB values based on acceleration using the following line:

`cp.pixels.fill(((R + abs(int(x))), (G + abs(int(y))), (B + abs(int(z)))))`

This involves some special math to work. Let's take a look!

First we'll look at the red value. We start with `R` which we created at the beginning of our loop. We're going to add the `x` value to `R`. However, there's a lot about the basic acceleration value that won't work for adding to color values, such as it potentially being a decimal or negative number. Luckily, Python has some easy ways to deal with this.

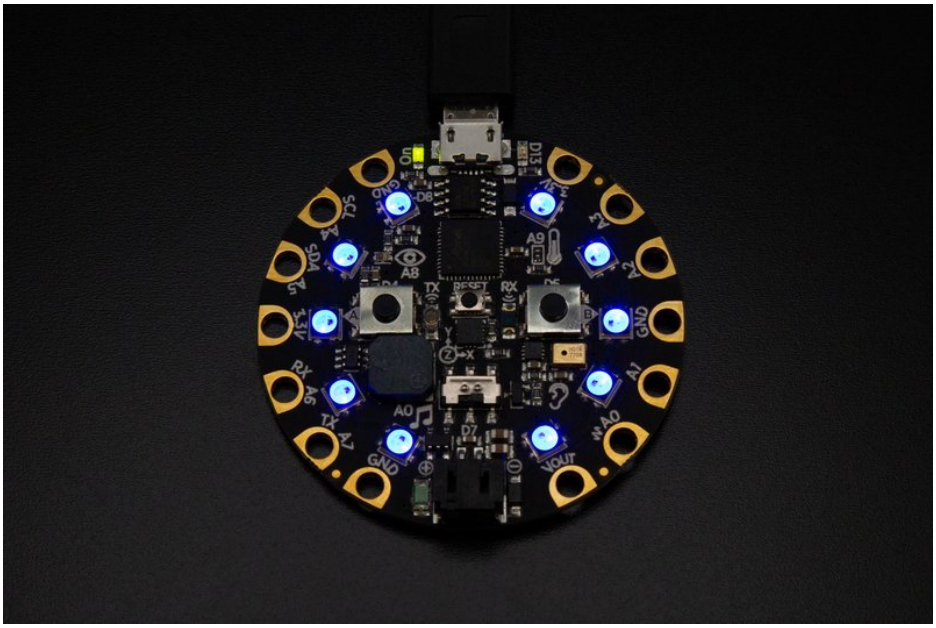
You'll notice that our value of x is modified a little with `abs(int(x))`. This returns the absolute value of the whole number value of `x`. Absolute values are explained here (https://adafru.it/BnD). Since color values are all whole numbers, we use `int(x)` to return only the nearest whole number value of `x`, instead of a long decimal which is often what acceleration returns. Since color values are all positive, we take the absolute value of `int(x)` to remove any potential negative numbers from the mix.

We add `abs(int(x))` to `R` and we have our `R` value to use for red! Then we do the same thing for `y` and `z`, except

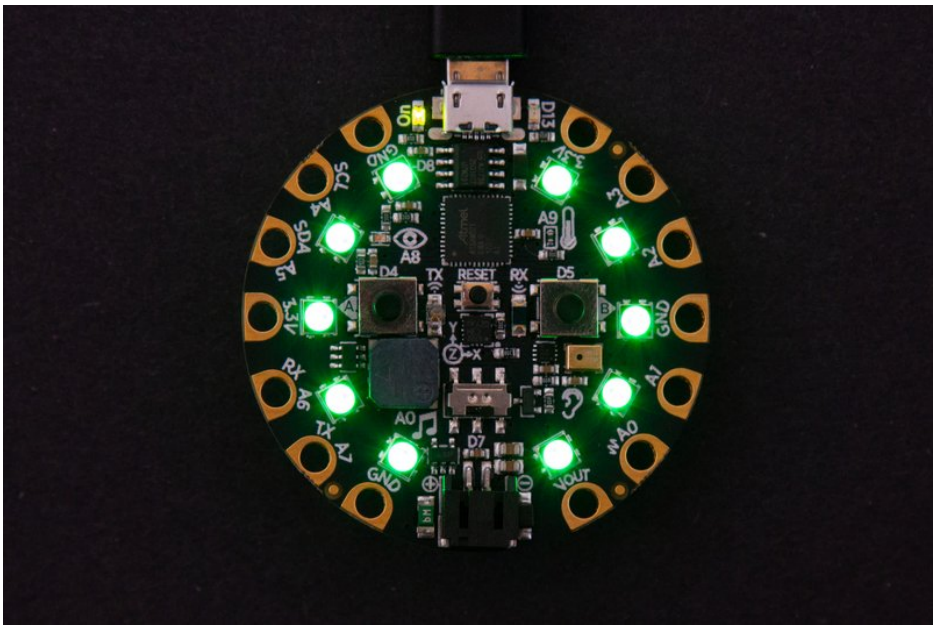
`abs(int(y))` is added to G and `abs(int(z))` is added to B. This gives us our three color values!

As you move the board around, the acceleration values change, and that causes each of our color values to be different. Now, depending on what angle you hold the board, you'll get a different color combination!

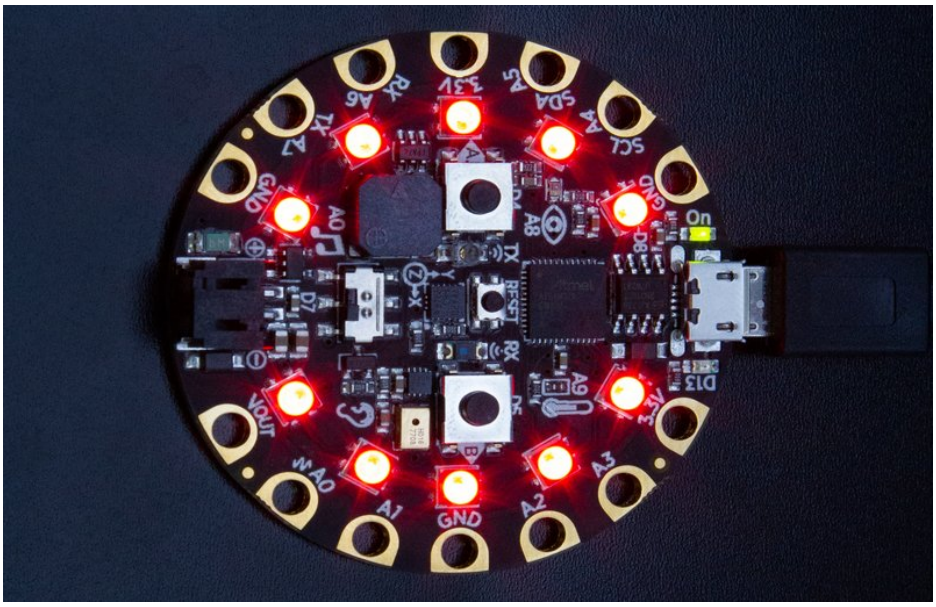
Remember the earlier example, where we explained that if the board is laying flat, the returned values are (0, 0, 9.8). This means, if the board is laying flat, facing up, while this code is running, the color values are `(0, 0, 9.8)`. So, you'll see if it's laying flat on your desk, it's blue!



If you hold it so the USB cable is on the top and pointed downwards, the values are, `(0, 9.8, 0)`, so the LEDs are green.



If you hold it so the USB cable is sideways, pointing left or right, the values are `(9.8, 0, 0)` so the LEDs are red.

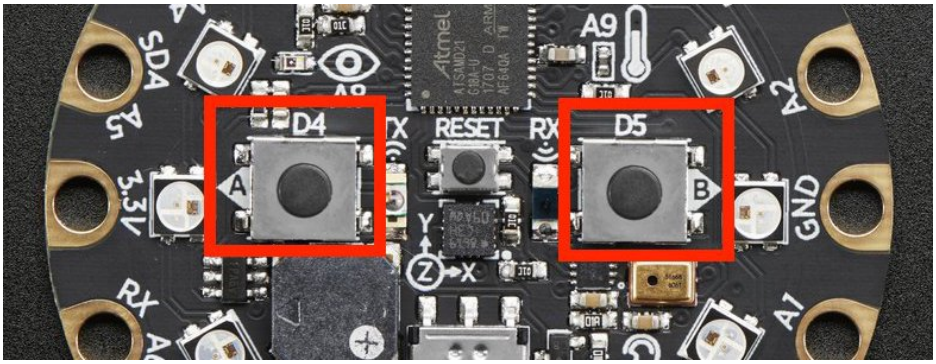


As you move the board around at different angles, you'll find every color between!

We also explained that if you shake the board, you'll get back higher values from the accelerometer. This means that the LEDs will be brighter if you shake it. Give it a try!

Buttons

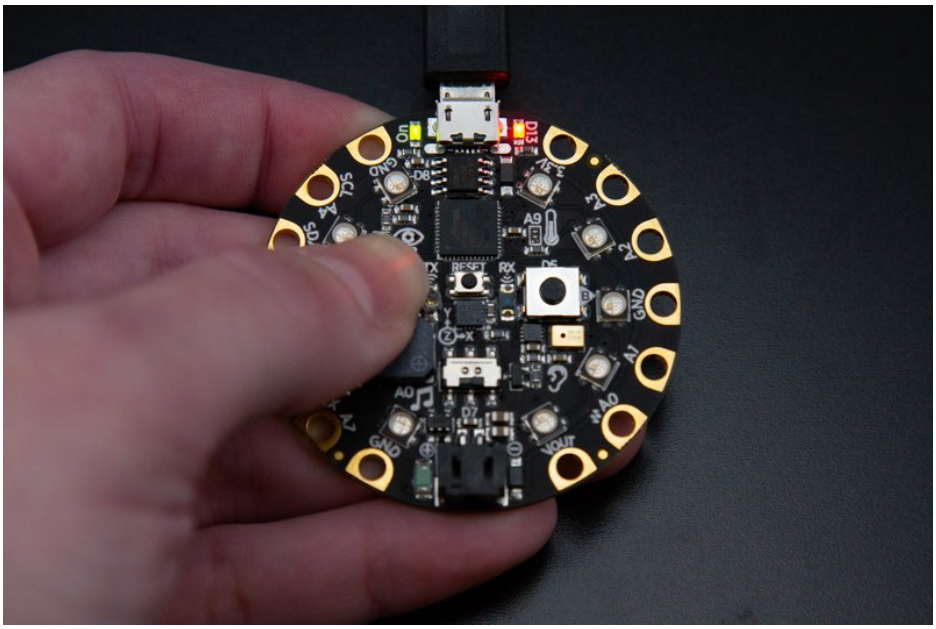
The Circuit Playground Express and Bluefruit have two buttons. Button A is on the left and button B is on the right. Though the images are of the Circuit Playground Express, the buttons are in the same location on the Bluefruit. These buttons can be used as inputs, which means you can use them to tell your board to do something when you press them.



Let's start with button A. Add the following code to your **code.py**. Remember, if you need help with this, check here (https://adafru.it/C96).

```
"""This example turns on the little red LED when button A is pressed."""
from adafruit_circuitplayground import cp

while True:
    if cp.button_a:
        print("Button A pressed!")
        cp.red_led = True
```

Now, press button A. Red LED!



Let's look at the code. First, we import `cp`.

Inside our loop, we check to see if button A is pressed with `if cp.button_a:`. Then, if it is, we print `Button A pressed!` to the serial console and we turn on the red LED!

Notice the LED stays on once button A is pressed. This is because we didn't tell the code to turn it off. So, let's try something a little different.

Add the following code to your **code.py**:

```
"""This example turns the little red LED on only while button B is currently being pressed."""
from adafruit_circuitplayground import cp

# This code is written to be readable versus being Pylint compliant.
# pylint: disable=simplifiable-if-statement

while True:
    if cp.button_b:
        cp.red_led = True
    else:
        cp.red_led = False

# Can also be written as:
#    cp.red_led = cp.button_b
```

Now press button B. Red LED! But only while it's pressed. Nice!

Let's take a look at the code. First we import `cp`.

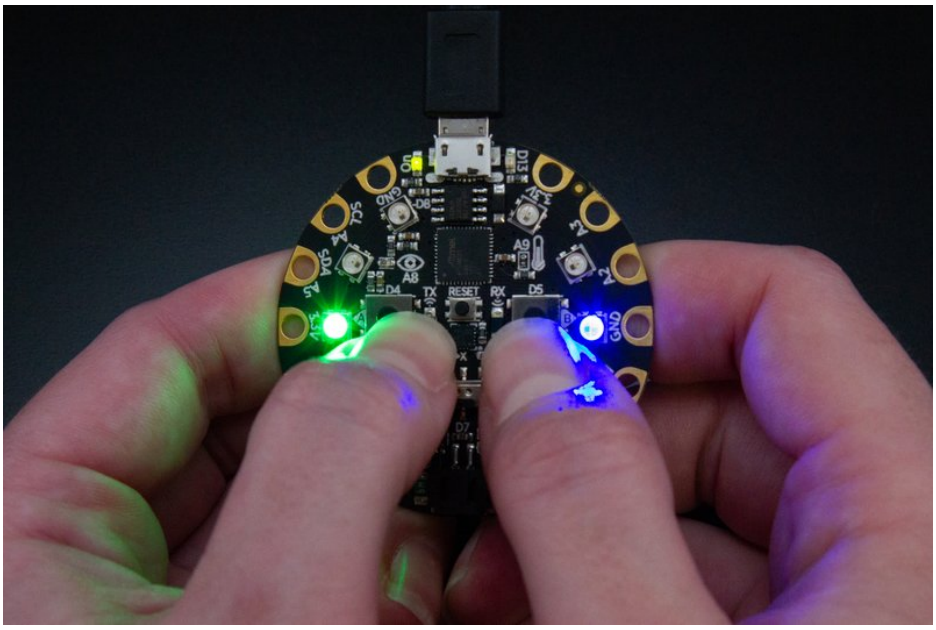
Inside our loop, we check to see if button B is pressed with `if cp.button_b:`. If it is, we turn on the red LED. Then, with our `else:`, we're telling the code, "otherwise, turn off the red LED." So, when the button is not being pressed, the LED turns off!

You can use both buttons in the same program. Let's change things up.

Add the following code to your **code.py**:

```
"""This example lights up the third NeoPixel while button A is being pressed, and lights up the
eighth NeoPixel while button B is being pressed."""
from adafruit_circuitplayground import cp

cp.pixels.brightness = 0.3
cp.pixels.fill((0, 0, 0))  # Turn off the NeoPixels if they're on!

while True:
    if cp.button_a:
        cp.pixels[2] = (0, 255, 0)
    else:
        cp.pixels[2] = (0, 0, 0)

    if cp.button_b:
        cp.pixels[7] = (0, 0, 255)
    else:
        cp.pixels[7] = (0, 0, 0)
```

Now press button A or B. Or press them both at the same time. Green and blue NeoPixels!



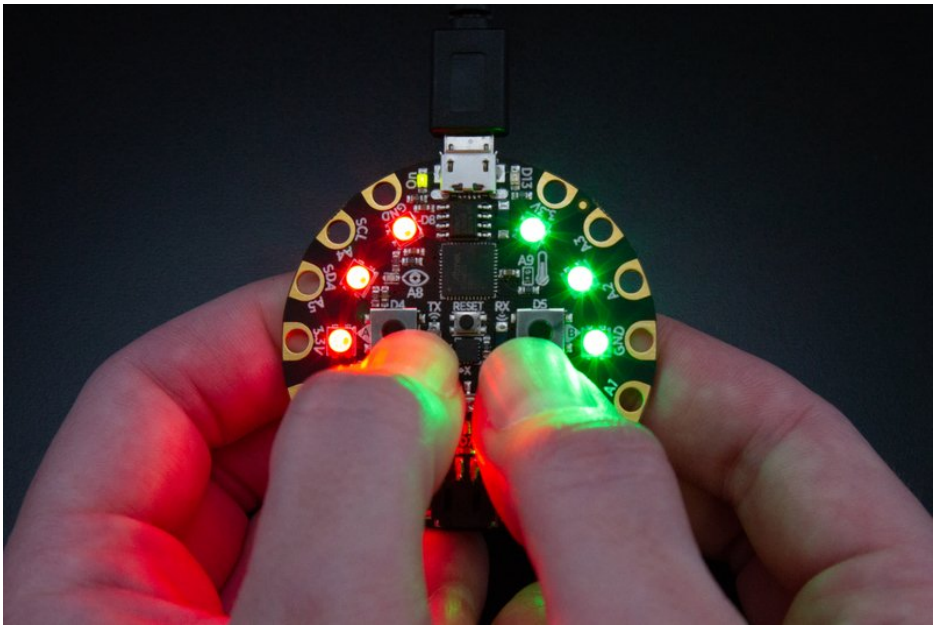
Our code is checking to see `if` each button is pressed. If it is, it turns on the LED next to the button to the specified color. Button A turns the LED next to it green. Button B turns the LED next to it blue. And, if the buttons are not being pressed, the LEDs are otherwise turned off by `cp.pixels.fill((0, 0, 0))`.

Half and Half

Let's get a little fancier. Add the following code to your code.py:

```
"""This example lights up half the NeoPixels red while button A is being pressed, and half the
NeoPixels green while button B is being pressed."""
from adafruit_circuitplayground import cp

cp.pixels.brightness = 0.3
cp.pixels.fill((0, 0, 0))  # Turn off the NeoPixels if they're on!

while True:
    if cp.button_a:
        cp.pixels[0:5] = [(255, 0, 0)] * 5
    else:
        cp.pixels[0:5] = [(0, 0, 0)] * 5

    if cp.button_b:
        cp.pixels[5:10] = [(0, 255, 0)] * 5
    else:
        cp.pixels[5:10] = [(0, 0, 0)] * 5
```

Now press button A or button B. Neopixels half and half, split down the middle, matching the sides the buttons are on!



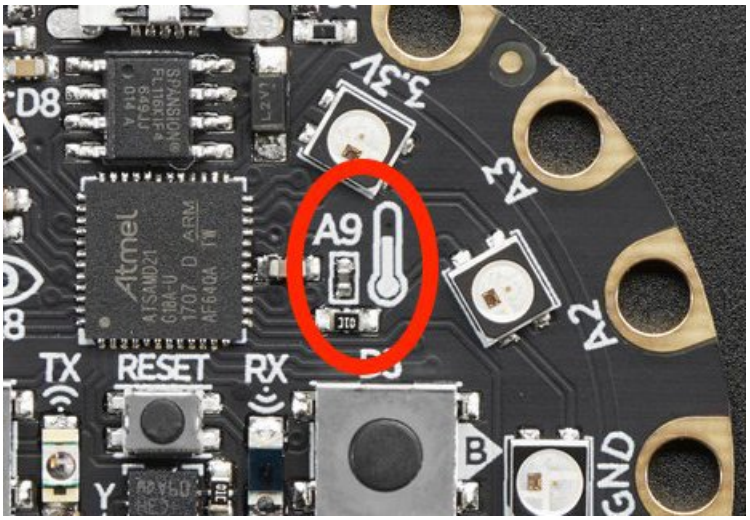
Here we're using a concept called slicing. Slicing allows you to specify a start point and an end point and enables us to tell the code to light up everything in between. So, instead of specifying a single LED with `[0]`, we tell the board to light up the first half of the LEDs on pressing button A with `cp.pixels[0:5] = [(255, 0, 0)] * 5`. The `[0:5]` is the start and end point, and the `* 5` is the slice size (5 out of 10 LEDs). We do the same with button B and the second half of the LEDs with `cp.pixels[5:10]`. And we tell the LEDs to otherwise be off if no buttons are pressed.

Note that the end points are 1 higher than the normal LED numbering - slice math is a little bit different than CircuitPython counting. Try playing with it a little bit. Change the first set to `cp.pixels[1:4] = [(255, 0, 0)] * 3`. See which LEDs light up!

If you try to specify a set of LEDs that's different from the slice size, you code won't run and an error will be printed to the serial console. For example, `cp.pixels[1:4] = [(255, 0, 0)] * 4` will fail because your slice size should be 3. So be sure to match them up!

Temperature

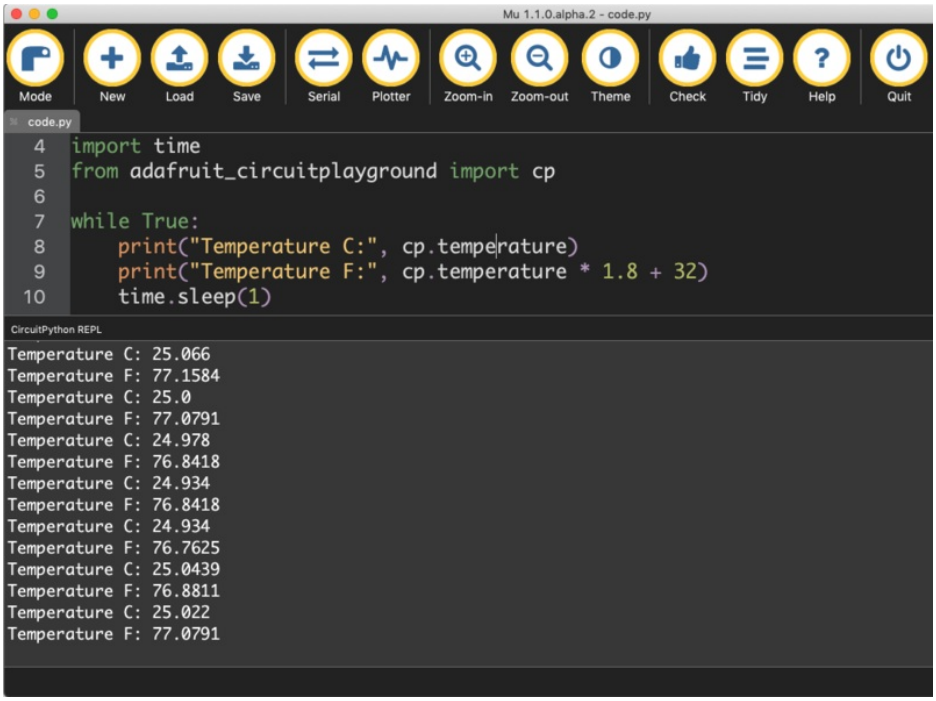
The Circuit Playground Express and Bluefruit have a temperature sensor built in, next to the little thermometer printed on the board. Though the images are of the Circuit Playground Express, the sensor is in essentially the same location on the Bluefruit. It's near the A9 label on the board. It returns the temperature in Celsius.



Add the following code to your **code.py**. Remember, if you need help with this, check here (https://adafru.it/C96).

```
"""This example uses the temperature sensor on the Circuit Playground, located next to the image of
a thermometer on the board. It prints the temperature in both C and F to the serial console. Try
putting your finger over the sensor to see the numbers change!"""
import time
from adafruit_circuitplayground import cp

while True:
    print("Temperature C:", cp.temperature)
    print("Temperature F:", cp.temperature * 1.8 + 32)
    time.sleep(1)
```

Open the serial console to see the temperature printed out. Try holding your finger over the thermometer printed on the board to see the values change!



Let's take a look at the code. We import `time` and `cp`.

Inside our loop, we print `Temperature C:`, followed by the temperature value, `cp.temperature`. This prints the temperature in Celsius.

But what if you're used to the temperature in Fahrenheit? It's as easy as a little math to display that as well. After printing the temp in C, we print `Temperature F:`, followed by `cp.temperature` again, this time modified by `* 1.8 + 32`, to convert it to Fahrenheit.

Then we have a `time.sleep(1)` to slow down the readings. If they're too fast, they're hard to read!

Plotting Temperature

Let's take a look at these values on the Mu plotter! Add the following code to your **code.py**:

```
"""If you're using Mu, this example will plot the temperature in C and F on the plotter! Click
"Plotter" to open it, and place your finger over the sensor to see the numbers change. The
sensor is located next to the picture of the thermometer on the CPX."""
import time
from adafruit_circuitplayground import cp

while True:
    print("Temperature C:", cp.temperature)
    print("Temperature F:", cp.temperature * 1.8 + 32)
    print((cp.temperature, cp.temperature * 1.8 + 32))
    time.sleep(0.1)
```

The code is almost identical, but we've added one line: `print((cp.temperature, cp.temperature * 1.8 + 32))`.

Note that the Mu plotter looks for **tuple** values to plot. Tuples in Python come in parentheses `()` with comma

separators. If you have two values, a tuple would look like `(1.0, 3.14)` - note the parentheses *around* the number set, and the *comma* between. That's why there's an extra set of parenthesis around and a comma between the two temperature values in `print((cp.temperature, cp.temperature * 1.8 +32))`.

As well, the Mu plotter requires that the tuple value be on a line all its own. That's why we can't simply add extra parenthesis and a comma to the `print("Temperature C:", cp.temperature)` line. The plotter doesn't know what to do with it if there's other information in there.

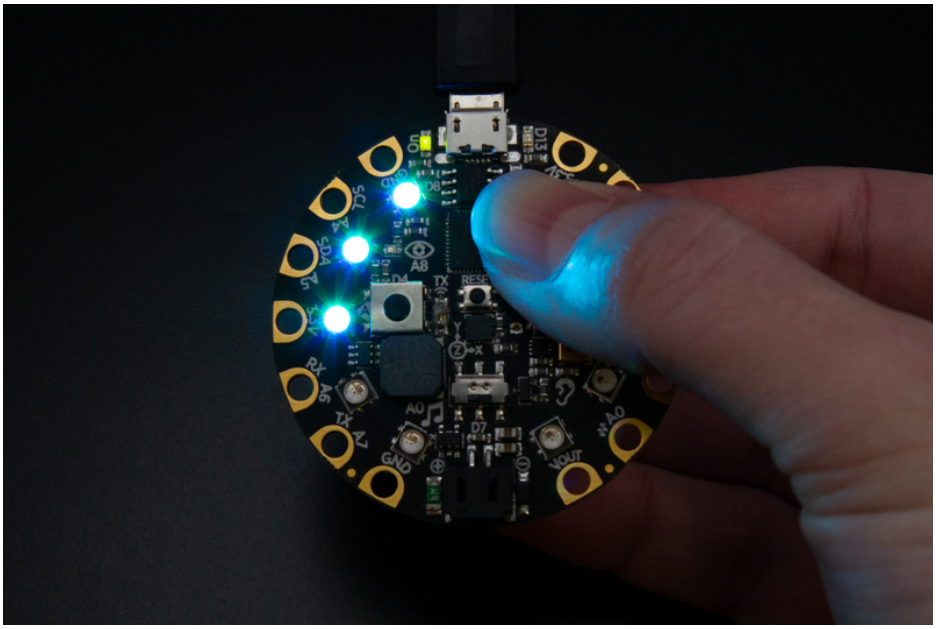
Click on the Plotter button on the top of Mu to see the plotter. Try breathing on your Circuit Playground to watch the plotter go up. Try setting it on an ice pack to watch the plotter go down!

Temperature Meter

You can also use the temperature values to create a fun light meter using the NeoPixels on your Circuit Playground! Add the following code to your **code.py**:

```
"""
This example use the temperature sensor on the Circuit Playground, located next to the picture of
the thermometer on the board. Try warming up the board to watch the number of NeoPixels lit up
increase, or cooling it down to see the number decrease. You can set the min and max temperatures
to make it more or less sensitive to temperature changes.
"""
import time
from adafruit_circuitplayground import cp

cp.pixels.auto_write = False
cp.pixels.brightness = 0.3

# Set these based on your ambient temperature in Celsius for best results!
minimum_temp = 24
maximum_temp = 30


def scale_range(value):
    """Scale a value from the range of minimum_temp to maximum_temp (temperature range) to 0-10
    (the number of NeoPixels). Allows remapping temperature value to pixel position."""
    return int((value - minimum_temp) / (maximum_temp - minimum_temp) * 10)


while True:
    peak = scale_range(cp.temperature)
    print(cp.temperature)
    print(int(peak))

    for i in range(10):
        if i <= peak:
            cp.pixels[i] = (0, 255, 255)
        else:
            cp.pixels[i] = (0, 0, 0)
    cp.pixels.show()
    time.sleep(0.05)
```

Now try holding your finger over the thermometer printed on your Circuit Playground and watch the LEDs light up! Remove your finger to watch the number of LEDs lit up change.



Let's take a look at the code. First we import `time`, and `cp`.

Next, we set `cp.pixels.auto_write = False`. This means that anything we tell the LEDs to do will not happen automatically. By default, this is set to `True`. This means, we tell the LEDs to turn on, and they turn on. If it's set to `False`, it means we have to include `cp.pixels.show()` after anything we try to tell the LEDs to do. This is required for this code to work since the LEDs turn on based on the temperature values.

We set the `brightness` to `0.3`, or 30%.

You should be able to see what the temperature changes are from when the Circuit Playground is simply sitting on your desk and when you're holding your finger over it. For best results, change the `minimum_temp` and `maximum_temp` to fit your ambient temperature values. Otherwise, you might not get the best results from the temperature meter. When sitting here, the minimum was about 24 degrees, and when holding a finger on it, the maximum was about 30. This is how we chose the values already in the code.

Next we have a helper function called `scale_range`. The temperature range is currently 24-30 but there are 10 NeoPixels. So, we include a helper function that scales the 24-30 range to 0-9 so we can map light levels to pixel position.

Our loop begins with setting `peak = scale_range(cp.temperature)`. Then we print the `cp.temperature` values and the `peak` values.

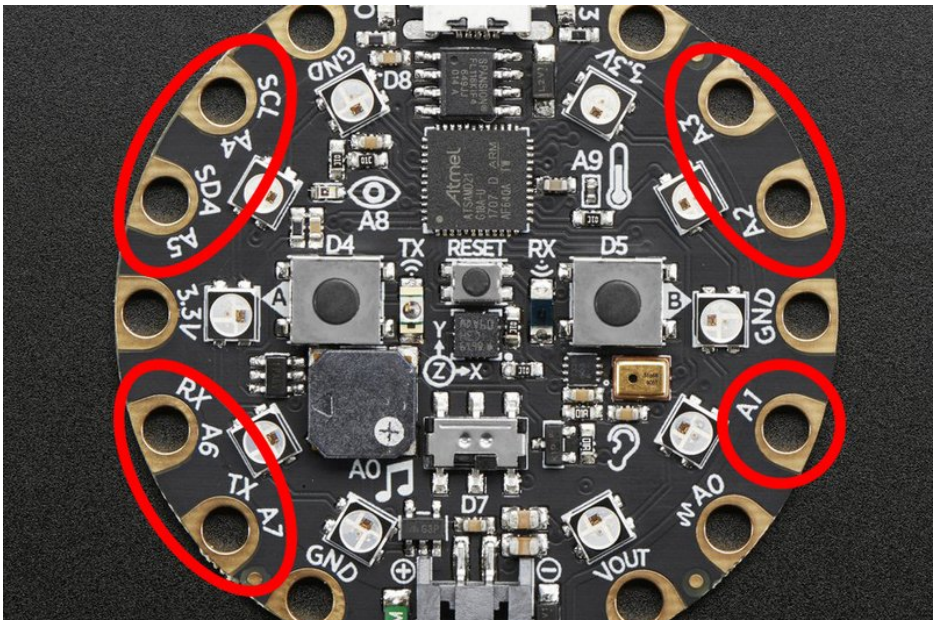
The next section takes the `peak` value and says for the total number of LEDs, whatever number `peak` is equal to or less than, light up that many LEDs, and otherwise turn them off. So, if peak is 4, light up 4 LEDs!

Then we have `cp.pixels.show()` to make the LEDs light up. And a `time.sleep(0.05)` to create a little delay.

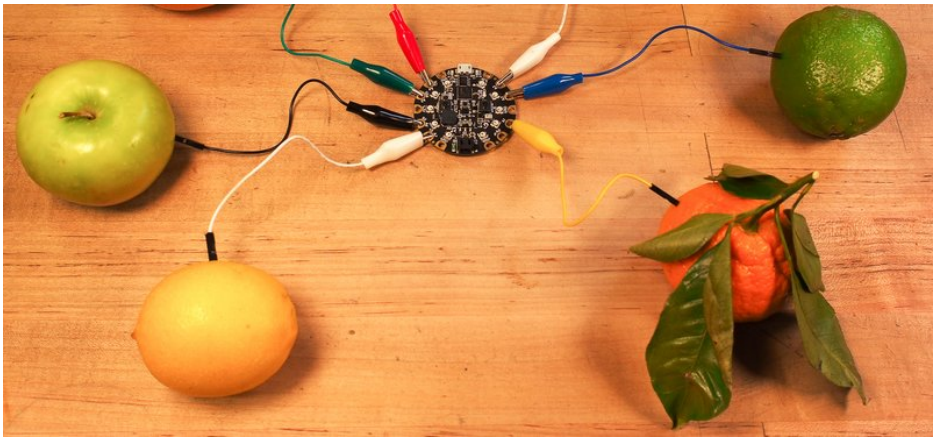
You can change the number values in `cp.pixels[i] = (0, 255, 255)` to change the color of the temperature meter. Give it a try!

Capacitive Touch

The Circuit Playground Express and Bluefruit have seven capacitive touch pads around the outside, labeled A1 - A6 and TX. Though the images are of the Circuit Playground Express, the touch pads are in the same location on the Bluefruit. These pads return True if you touch them. So you can use them as inputs to do all sorts of fun stuff!



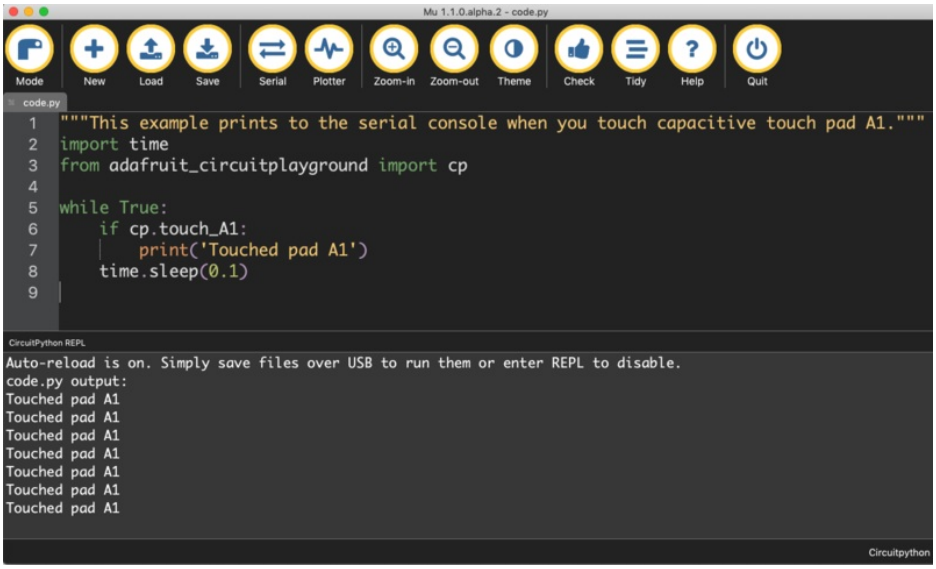
Since the pads are capacitive, you can also attach alligator clips to them and any number of capacitive items and touch those to activate them as well! For example, you could attach one end of an alligator clip to one of the pads and the other end to an apple or a lime. Or place the other end in a glass of water. Then touch the fruit or the glass of water. You'll activate the pad!



Add the following code to your **code.py**. Remember, if you need help with this, check here (https://adafru.it/C96).

```
"""This example prints to the serial console when you touch capacitive touch pad A1."""
from adafruit_circuitplayground import cp

while True:
    if cp.touch_A1:
        print("Touched pad A1")
```

Open the serial console. Now, touch the pad labeled A1 on your Circuit Playground. `Touched pad A1`!



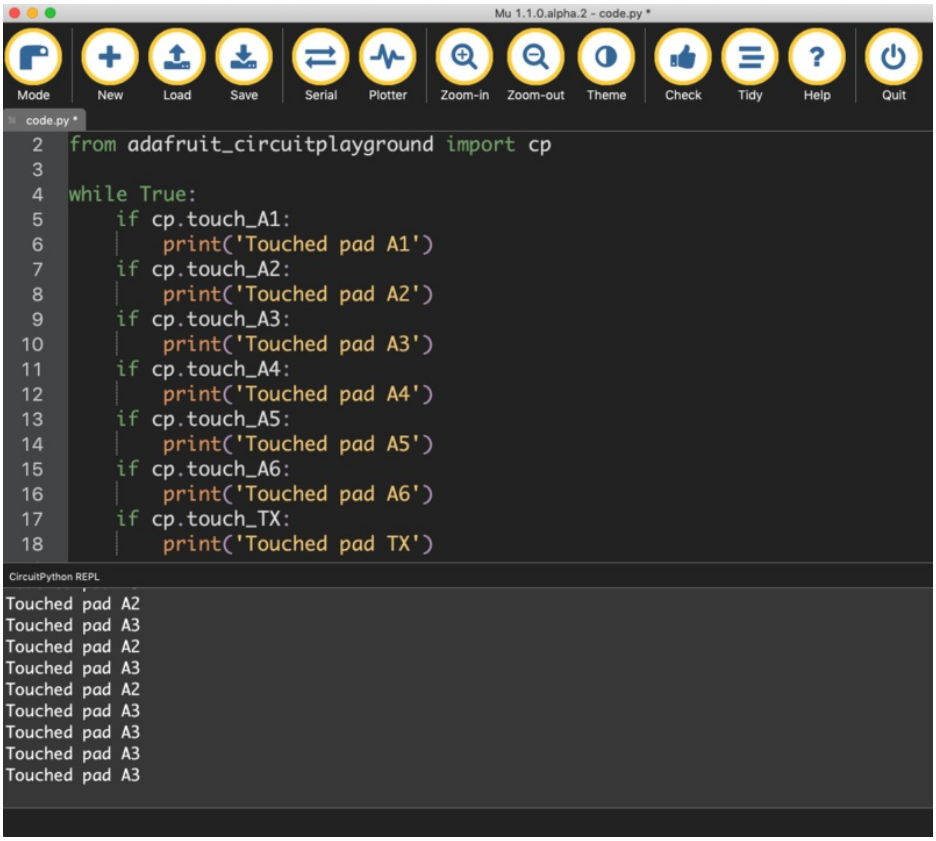
Let's look at the code. First we import `time` and `cp`.

Inside our loop, we check to see if pad A1 is touched with `if cp.touch_A1:`. If it is, we `print` `Touched pad A1` to the serial console. Then we have a `time.sleep(0.1)` to slow down the speed of the printing.

Nice! But what about the rest of the touch pads? Add the following code to your **code.py**.

```
"""This example prints to the serial console when you touch the capacitive touch pads."""
from adafruit_circuitplayground import cp

while True:
    if cp.touch_A1:
        print("Touched pad A1")
    if cp.touch_A2:
        print("Touched pad A2")
    if cp.touch_A3:
        print("Touched pad A3")
    if cp.touch_A4:
        print("Touched pad A4")
    if cp.touch_A5:
        print("Touched pad A5")
    if cp.touch_A6:
        print("Touched pad A6")
    if cp.touch_TX:
        print("Touched pad TX")
```

Now look at the serial console and touch any of the touch pads. `Touched pad...`!



The code begins the same way. But, we've added in another two lines for each touch pad. We check `if` each pad is touched, and if it is, we print `Touched pad` and the pad number to the serial console.

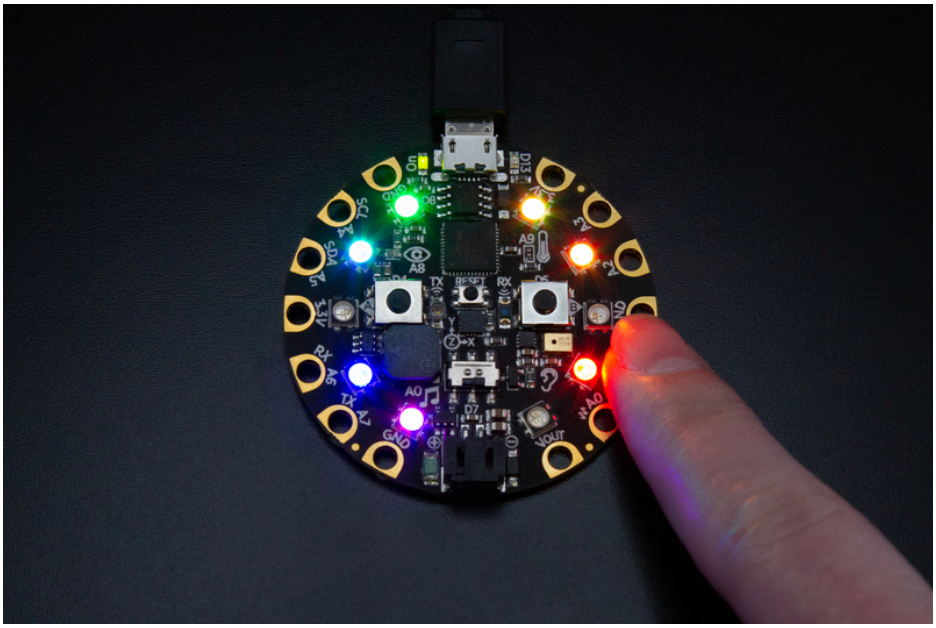
Now we've included all of the touch pads. Let's do something with them!

Touch the Rainbow

Add the following code to your **code.py**.

```
"""This example uses the capacitive touch pads on the Circuit Playground. They are located around
the outer edge of the board and are labeled A1-A6 and TX. (A0 is not a touch pad.) This example
lights up the nearest NeoPixel to that pad a different color of the rainbow!"""
import time
from adafruit_circuitplayground import cp

cp.pixels.brightness = 0.3

while True:
    if cp.touch_A1:
        print("Touched A1!")
        cp.pixels[6] = (255, 0, 0)
    if cp.touch_A2:
        print("Touched A2!")
        cp.pixels[8] = (210, 45, 0)
    if cp.touch_A3:
        print("Touched A3!")
        cp.pixels[9] = (155, 100, 0)
    if cp.touch_A4:
        print("Touched A4!")
        cp.pixels[0] = (0, 255, 0)
    if cp.touch_A5:
        print("Touched A5!")
        cp.pixels[1] = (0, 135, 125)
    if cp.touch_A6:
        print("Touched A6!")
        cp.pixels[3] = (0, 0, 255)
    if cp.touch_TX:
        print("Touched TX!")
        cp.pixels[4] = (100, 0, 155)
    time.sleep(0.1)
```

Now touch each touch pad. You get an LED in one color of the rainbow for each of them!



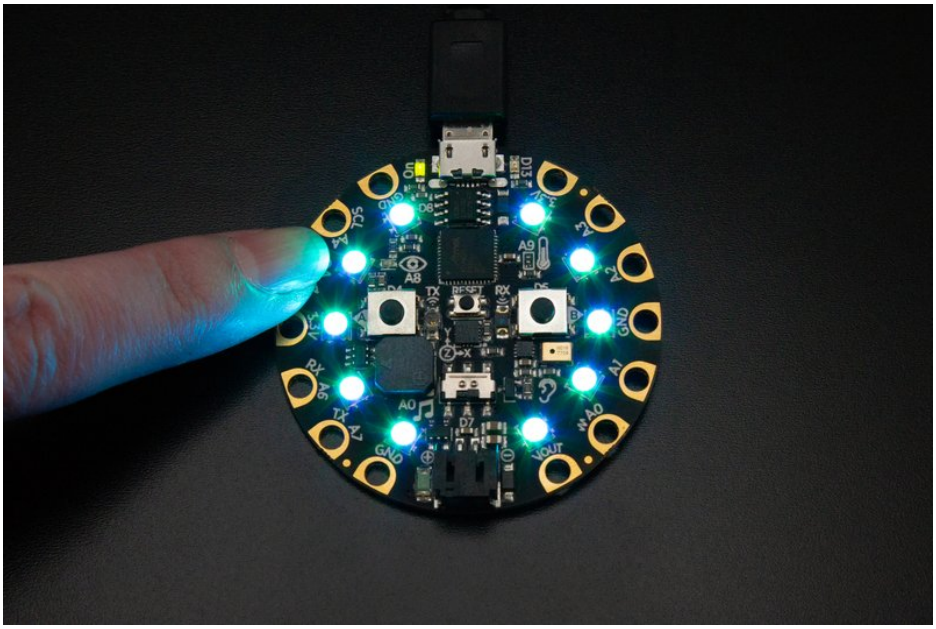
Now let's look at the code. We import `time` and `cp`. We set the LED brightness to 30%. We check to see `if` each pad is touched, and if it is, we `print` to the serial console. This time, though, we also light up a specific LED with each pad

using `cp.pixels[#] = (r, g, b)` where `#` is the pixel number and `r, g, b` are the color values. We didn't include any code to tell the LEDs to turn off, so they will stay on once you turn them on.

Now let's add more light! Add the following code to your **code.py**.

```
"""This example uses the capacitive touch pads on the Circuit Playground. They are located around
the outer edge of the board and are labeled A1-A6 and TX. (A0 is not a touch pad.) This example
lights up all the NeoPixels a different color of the rainbow for each pad touched!"""
import time
from adafruit_circuitplayground import cp

cp.pixels.brightness = 0.3

while True:
    if cp.touch_A1:
        print("Touched A1!")
        cp.pixels.fill((255, 0, 0))
    if cp.touch_A2:
        print("Touched A2!")
        cp.pixels.fill((210, 45, 0))
    if cp.touch_A3:
        print("Touched A3!")
        cp.pixels.fill((155, 100, 0))
    if cp.touch_A4:
        print("Touched A4!")
        cp.pixels.fill((0, 255, 0))
    if cp.touch_A5:
        print("Touched A5!")
        cp.pixels.fill((0, 135, 125))
    if cp.touch_A6:
        print("Touched A6!")
        cp.pixels.fill((0, 0, 255))
    if cp.touch_TX:
        print("Touched TX!")
        cp.pixels.fill((100, 0, 155))
    time.sleep(0.1)
```

Touch each pad. You get every LED lit up in one color of the rainbow for each of them!

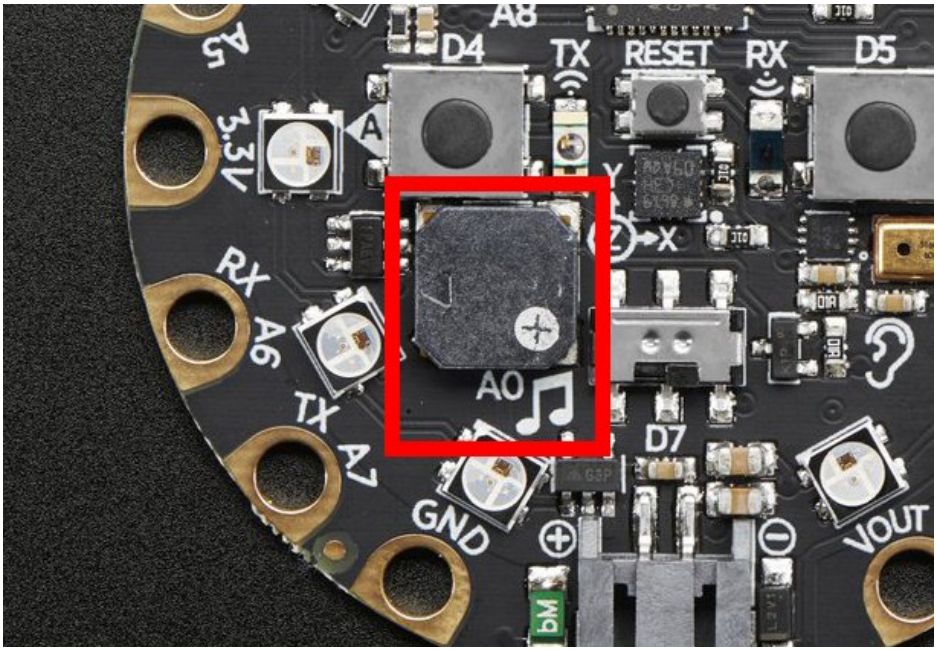


The code is almost identical, except instead of `cp.pixels[#]`, we use `cp.pixels.fill((r, g, b))` to light up every LED instead of only one.

You can change the color values to whatever you like to create your own personal rainbow. Give it a try!

Play Tone

The Circuit Playground Express and Bluefruit have a built-in speaker above the music note printed on the board. It is the grey box with a + on it, below button A, to the left of the slide switch. Though the image is of the Circuit Playground Express, the speaker is in the same location on the Bluefruit. This speaker is capable of multiple things including the ability to play tones.



Add the following code to your **code.py**. Remember, if you need help with this, check here (https://adafru.it/C96).

```
"""This example plays two tones for 1 second each. Note that the tones are not in a loop - this is
to prevent them from playing indefinitely!"""
from adafruit_circuitplayground import cp

cp.play_tone(262, 1)
cp.play_tone(294, 1)
```

When you save the code, you'll have two tones!

First we import `cp`. Then, we play one tone, followed by another with `cp.play_tone(262, 1)` and `cp.play_tone(294, 1)`.

Note that we did not include a loop in this code. This is because if the code is in a loop, it will continue playing indefinitely. This is not always desirable, so we've designed the code to play each tone once.

`cp.play_tone()` requires two things from you: a frequency in hertz and a length of time in seconds. So anytime you want to use it, you'll add `cp.play_tone(frequency, seconds)` to your code, where `frequency` is the hertz of the tone you'd like to play, and `seconds` is the length of time you'd like it to play.

There are many tone generators available on the internet that will give you the hertz of a specific tone. The two tones we've added to the current code are middle C and the D above middle C. Try adding another tone. Have fun with it!

Two Tone Buttons

You can use any of the inputs that we've talked about to play tones. Let's try using the buttons. Add the following code to your **code.py**.

```
"""This example plays a different tone for a duration of 1 second for each button pressed."""
from adafruit_circuitplayground import cp

while True:
    if cp.button_a:
        cp.play_tone(262, 1)
    if cp.button_b:
        cp.play_tone(294, 1)
```

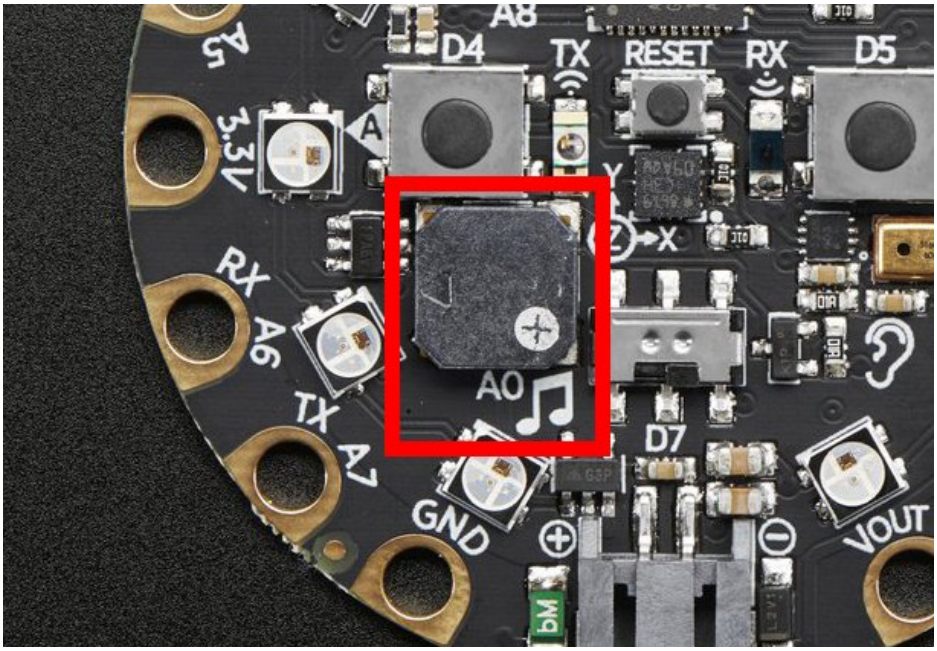
Now, press each button. Each one plays a tone for one second!

This code is the same as previous code using the buttons. Inside the loop, it checks to see `if` each button is pressed. This time, if button A is pressed, it plays a `262` Hz tone for `1` second, and if button b is pressed, it plays a `294` Hz tone for `1` second.

You can use any of the inputs we've discussed in this guide to trigger a tone. Try replacing the button presses with touch pads. Have fun with it!

Start and Stop Tone

The Circuit Playground Express and Bluefruit have a built-in speaker above the music note printed on the board. It is the grey box with a + on it, below button A, to the left of the slide switch. Though the image is of the Circuit Playground Express, the speaker is in the same location on the Bluefruit. This speaker is capable of multiple things including the ability to play tones.



What if, instead of playing a tone for a specified amount of time (using `play_tone()`), you want to play the tone only when you provide an input? For example, instead of playing a tone for 1 second, what if you want the tone to play while you're pressing a button? Or touching a touch pad? You can do that!

Add the following code to your **code.py**. Remember, if you need help with this, check here (https://adafru.it/C96).

```
"""This example plays a different tone for each button, while the button is pressed."""
from adafruit_circuitplayground import cp

while True:
    if cp.button_a:
        cp.start_tone(262)
    elif cp.button_b:
        cp.start_tone(294)
    else:
        cp.stop_tone()
```

Press button A. Now, press button B. Each button plays a tone, but only while it's being pressed!

Let's look at the code. First we import `cp`.

Inside our loop, we check to see `if` the buttons are being pressed. If button A is pressed, we start a tone with `cp.start_tone(262)`. If button B is pressed, we start a tone with `cp.start_tone(294)`. Otherwise, if they're not being pressed, we stop the tone. That's it!

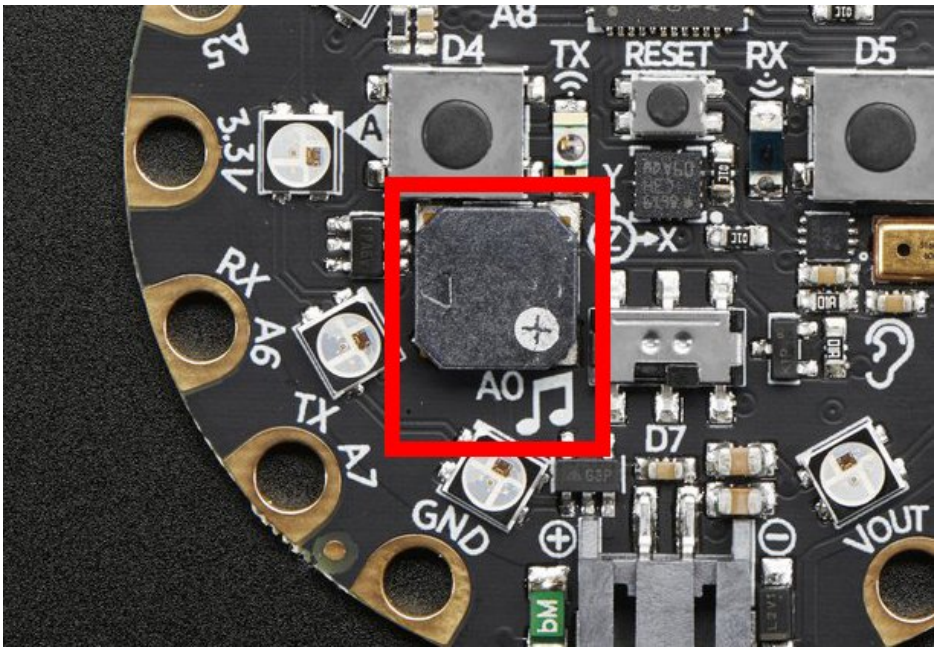
`cp.start_tone()` requires one thing from you, a frequency in hertz of the tone you would like to start. So anytime you want to use it, you'll add `cp.start_tone(frequency)` to your code, where `frequency` is the hertz of the tone you'd like to start.

`cp.start_tone()` requires `cp.stop_tone()` to stop playing. Without it, you'll start the tone and it will play indefinitely. You'll know very quickly if you've forgotten to add `cp.stop_tone()`!

Try replacing buttons A and B with touch pads A1 and A2, and change the frequencies to have different tones. Try using all the touch inputs to have more tone options!

Play File

The Circuit Playground Express and Bluefruit have a built-in speaker above the music note printed on the board. It is the grey box with a + on it, below button A, to the left of the slide switch. Though the image is of the Circuit Playground Express, the speaker is in the same location on the Bluefruit. The speaker is also able to play monotone music encoded in a special format called wav files!



Sound files for the Circuit Playground library should be 22,050 kHz, 16-bit, mono (or less) WAV files to play on these boards. If you have an MP3 or a file you downloaded and are unsure of the encoding, you can follow this audio conversion guide (https://adafru.it/BvU) to get your files into the proper format.

For testing, we've prepared two WAV files in the proper format. You can download the following two .wav files and copy them to your Circuit Playground **CIRCUITPY** drive.

https://adafru.it/I6B

https://adafru.it/I6B

https://adafru.it/I6C

https://adafru.it/I6C

Add the following code to your **code.py**. Remember, if you need help with this, check here (https://adafru.it/C96).

```
"""THIS EXAMPLE REQUIRES A WAV FILE FROM THE examples FOLDER IN THE
Adafruit_CircuitPython_CircuitPlayground REPO found at:
https://github.com/adafruit/Adafruit_CircuitPython_CircuitPlayground/tree/master/examples

Copy the "dip.wav" file to your CIRCUITPY drive.

Once the file is copied, this example plays a wav file!"""
from adafruit_circuitplayground import cp

cp.play_file("dip.wav")
```

Dip!

Let's look at the code. First we import `cp`.

Then, we play a wav file called "dip.wav" with `cp.play_file("dip.wav")`. That's it!

Note that we did not include a loop in this code. This is because if the code is in a loop, it will continue playing indefinitely. This is not always desirable, so we've designed the code to play the file once.

`cp.play_file()` requires one thing from you: the name of the wav file you're trying to play back in quotation marks. This is how it knows what file to play. So anytime you want to use it, you'll want to add `cp.play_file("Filename.wav")` to your code, replacing `Filename.wav` with the name of your wav file. It is case sensitive, so match the file name exactly.

Let's add some inputs and another wav file. Add the following code to your **code.py**.

```
"""THIS EXAMPLE REQUIRES A WAV FILE FROM THE examples FOLDER IN THE
Adafruit_CircuitPython_CircuitPlayground REPO found at:
https://github.com/adafruit/Adafruit_CircuitPython_CircuitPlayground/tree/master/examples

Copy the "dip.wav" and "rise.wav" files to your CIRCUITPY drive.

Once the files are copied, this example plays a different wav file for each button pressed!"""
from adafruit_circuitplayground import cp

while True:
    if cp.button_a:
        cp.play_file("dip.wav")
    if cp.button_b:
        cp.play_file("rise.wav")
```

Now press button A. Dip! Press button B. Rise!

Inside the loop, we check to see `if` each button is pressed. If button A is pressed, we play `"dip.wav"`. If button B is pressed, we play `"rise.wav"`.

Notice if you press button B and then immediately try to press button A, the rise.wav file completes before you're able to dip again. This is because you cannot begin playing another file until the first file is completed. So, if you have a really long wav file, you'll find you can't do anything else until the file is finished playing. Keep that in mind if you're going to include wav files with other code.

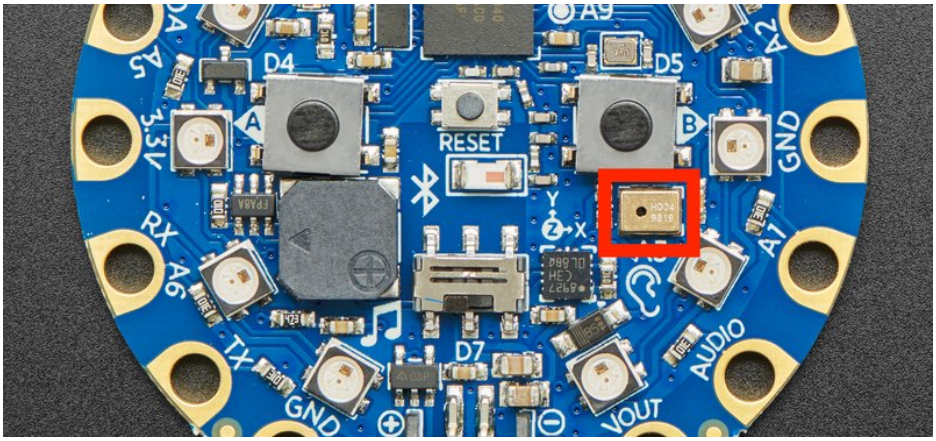
You can use any of the inputs we've discussed to trigger a file to play. Try replacing the button presses with touch inputs. Try adding different files to use!

If your code is running but your file doesn't sound quite right or doesn't play back, be sure to check the encoding of your sound file by following this Adafruit guide (https://adafru.it/BvU).

Sound

This Circuit Playground library feature is only available on the Circuit Playground Bluefruit.

The Circuit Playground Bluefruit has a sound sensor located on the right side of the board, above the ear printed on the board, and below button B. This sensor can be used to detect sound levels.

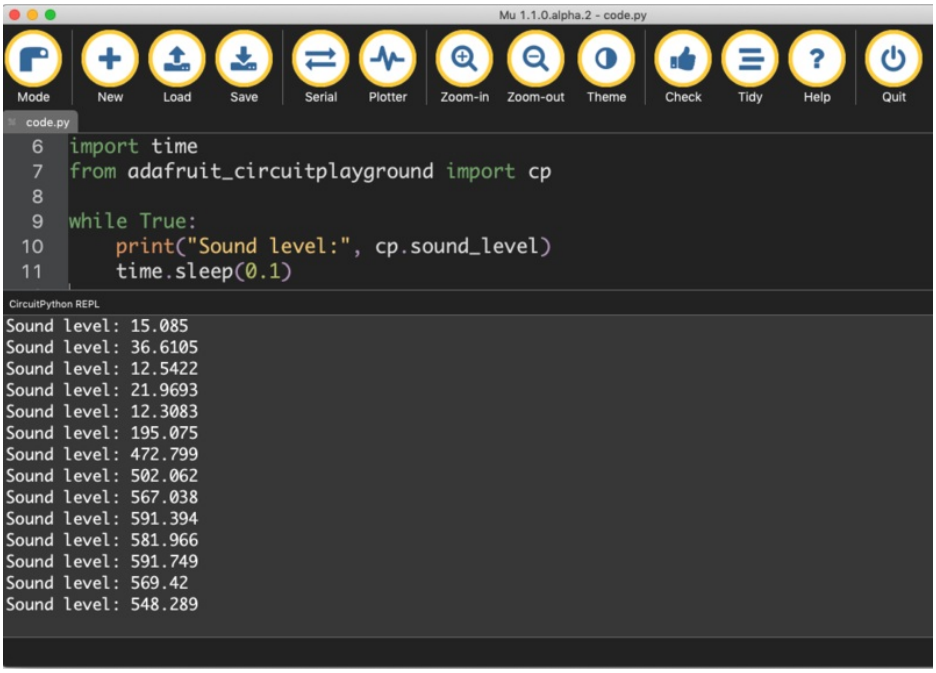


While the Circuit Playground Express also has a sound sensor, this feature of the Circuit Playground library is not available for the Express. The Express and Bluefruit have different microcontroller chips. The SAMD21 on the Express is not capable of handling the sound sensor features of the Circuit Playground Library.

Add the following code to your **code.py**. Remember, if you need help with this, check here (https://adafru.it/C96).

```
"""
This example prints out sound levels using the sound sensor on a Circuit Playground Bluefruit.
Try making sounds towards the board to see the values change.

NOTE: This example does NOT support Circuit Playground Express.
"""
import time
from adafruit_circuitplayground import cp

while True:
    print("Sound level:", cp.sound_level)
    time.sleep(0.1)
```

Open the serial console to see the sound level printed out. Try making noise at your Circuit Playground to see the values change!



Let's look at the code. First we import `time` and `cp`.

Inside our loop, we print to the serial console, `Sound level:` followed by the sound level value, `cp.sound_level`. Then we have a `time.sleep(0.1)` to slow down the speed at which it prints to the serial console. If it's too fast, it's difficult to read!

Plotting Sound Level

Let's take a look at these values on the Mu plotter! Add the following code to your **code.py**:

```
"""
This example prints out sound levels using the sound sensor on a Circuit Playground Bluefruit. If
you are using Mu, open the plotter to see the sound level plotted. Try making sounds towards the
board to see the values change.

NOTE: This example does NOT support Circuit Playground Express.
"""
import time
from adafruit_circuitplayground import cp

while True:
    print("Sound level:", cp.sound_level)
    print((cp.sound_level,))
    time.sleep(0.1)
```

The code is almost identical, but we've added one line, `print((cp.sound_level,))`.

Note that the Mu plotter looks for **tuple** values to plot. Tuples in Python come in parentheses `()` with comma separators. If you have two values, a tuple would look like `(1.0, 3.14)`. Since we have only one value, we need to have it print out like `(1.0,)` - note the parentheses *around* the number, and the *comma* after the number. Thus the extra parentheses and comma in `print((cp.sound_level,))`.

Click on the Plotter button on the top of Mu to see the plotter. Try making sounds towards your board to see the plotter line go up. Try being quiet to see the plotter line go down. Have fun with it!

Loud Sound

What if you wanted to use a sound as an input? With the `loud_sound()` feature, you can! It allows you to use a clap, snap or any other suitably loud sound as an input.

The following example lights up the NeoPixel LEDs when a loud enough sound occurs. Add the following code to your **code.py**:

```
"""
This example lights up the NeoPixels on a Circuit Playground Bluefruit in response to a loud sound.
Try snapping or clapping near the board to trigger the LEDs.

NOTE: This example does NOT support Circuit Playground Express.
"""
import time
from adafruit_circuitplayground import cp

while True:
    if cp.loud_sound():
        cp.pixels.fill((50, 0, 50))
        time.sleep(0.2)
    else:
        cp.pixels.fill((0, 0, 0))
```

Try clapping, snapping, or yelling at your board. Purple NeoPixels!

Let's take a look at the code. First we import `time` and `cp`.

Inside our loop, we begin by saying if a loud sound occurs, `if cp.loud_sound()`, turn on the NeoPixels a slightly dim purple, `cp.pixels.fill((50, 0, 50))`. Then we add a `time.sleep(0.2)` so the LEDs stay on long enough to see them. Without it, they only flash on for a moment.

Then we say, otherwise, turn the pixels off by setting them to `(0, 0, 0)`. Without this, the pixels would turn on and stay on.

Loud Sound Threshold

If you find it's too easy or too difficult to trigger the loud sound, you can decrease or increase the threshold. `loud_sound()` defaults to `sound_threshold=200`. To make it harder to trigger, you can increase the threshold. Add the following code to your **code.py**:

```
"""
This example lights up the NeoPixels on a Circuit Playground Bluefruit in response to a loud sound.
Try snapping or clapping near the board to trigger the LEDs.

NOTE: This example does NOT support Circuit Playground Express.
"""
import time
from adafruit_circuitplayground import cp

while True:
    if cp.loud_sound(sound_threshold=250):
        cp.pixels.fill((50, 0, 50))
        time.sleep(0.2)
    else:
        cp.pixels.fill((0, 0, 0))
```

The code is the same except we've increased the threshold by setting `sound_threshold=250`, making it require a louder sound to trigger.

If you find it's too difficult to trigger, you can lower the threshold, making it require a quieter sound to trigger. Try setting `sound_threshold=150` to see the difference.

Now you can use sound as an input on the Circuit Playground Bluefruit. Try combining it with the other concepts learned in this guide to see what else you can do!

Time to Get Creative!

Now that you have examples of how everything works, it's time to get creative! Try combining different concepts to put together a whole new project. You could make something like a capacitive touch combination lock or a book light. The possibilities are endless!

The following guides will only work on Circuit Playground Express. Though they use a different import and code format, they will still work with the current version of the Circuit Playground library for Circuit Playground Express.

Circuit Playground Express Project Guides Using the Circuit Playground Library:

- Circuit Playground Express: Piano in the Key of Lime (https://adafru.it/BeI)
- UFO Flying Saucer with Circuit Playground Express (https://adafru.it/BnG)
- CircuitPython Snow Globe (https://adafru.it/BnH)
- Hacking Ikea Lamps with Circuit Playground Express: CircuitPython Creature Friend (https://adafru.it/Bnt)
- Combo Dial Safe with Circuit Playground Express (https://adafru.it/BnE)
- Fruitbox Sequencer: Musically Delicious Step Pattern Generator (https://adafru.it/BnF)

The Technical Side

If you're new to programming, and looking for an easy way to get started with your Circuit Playground Express and CircuitPython, the important thing to know is that this library provides exactly that. However, if you'd like a deeper explanation of how it does that, we've got you covered. This section gets into some fairly technical concepts, so don't worry if you don't follow everything. We've included this to clear up any questions more advanced users may have about how the library works behind the scenes.

There are multiple layers to how this library functions. The following is an explanation of the Circuit Playground library.

Note: This library works with the Circuit Playground Express and Circuit Playground Bluefruit, NOT the Circuit Playground Classic. Any reference in this explanation to "Circuit Playground" is referring to the Express and Bluefruit only.

This section is not meant for beginners. It includes a very technical explanation of how the Circuit Playground library works. It assumes that you have a certain level of knowledge about CircuitPython, its underlying code, and how modules work.

Circuit Playground Library Modules

The library is divided up into multiple modules. The `circuit_playground_base` module defines a base class called `CircuitPlaygroundBase`, which includes the library features available for all of the Circuit Playground boards, such as `red_led`, `button_a`, etc. The `express` module defines the `Express` class, which is a subclass of `CircuitPlaygroundBase`, which adds features available for only the Circuit Playground Express, such as an alias for `touch.A7` to `touch.TX` (only the CPX has the A7 label on the seventh touch pad). The `bluefruit` module similarly defines the `Bluefruit` class, which adds features available for only the Circuit Playground Bluefruit, such as `sound_level` and `loud_sound`. The `Express` and `Bluefruit` classes inherit the features of the `CircuitPlaygroundBase` class so when either of the board-specific modules is imported, all of the base and board-specific features are made available.

Within the modules, all of the necessary libraries and CircuitPython modules are imported. All of the hardware and software initialisation is done in `__init__()` within the module, such as initialising the accelerometer or creating variables for later use. Then we use methods and properties to expose the features for use in your code.

Circuit Playground Library Use

To use the library, you include `from adafruit_circuitplayground import cp` at the beginning of your program. The first thing the library does is use `sys.platform` to determine whether the connected board is an **Atmel SAMD21** or an **nRF52840** microcontroller. This code is contained within the `__init__.py` file, and is run on import before any other code. Based on the results, it imports the appropriate library module, either `express` or `bluefruit`. So for instance if you are running on a Circuit Playground Bluefruit, all of the `bluefruit` features will be imported. This import mechanism allows the same piece of code to work with all Circuit Playground boards. Here is the essence of `__init__.py`:

```
import sys
if sys.platform == 'nRF52840':
    from .bluefruit import cpb as cp
elif sys.platform == 'Atmel SAMD21':
    from .express import cpx as cp
```

Once imported, all of features for the connected board are available for use as `cp.feature_name`. For example, to address the little red status LED, you would include `cp.red_led` in your program.

This library is unusual in that you don't create the primary object yourself. Instead, the object is created on import. When you do `from adafruit_circuitplayground import cp`, you're importing the `cp` object has already been created and assigned the name `cp`. You do not use the `Express` or `Bluefruit` class directly.

This library was originally written only for Circuit Playground Express. Previously, you would have used the import `from adafruit_circuitplayground.express import cpx`. `cpx` is the name for the `Express` class object created inside the `express.py` module. When we added support for Circuit Playground Bluefruit, we had the `bluefruit.py` module create an object named `cpb`, analogous to `cpx`. However, we realized that any code written to use both boards would have to have all its references to `cpb` change to `cpx` or *vice versa*. To alleviate this, we added `__init__.py`, which, as described above discovers which board is the code is running on, and imports either `cpx` or `cpb`, renaming it to just `cp`.

Circuit Playground Library vs. Basic CircuitPython

Without this library, each feature of the board would require setup in your code, ranging from one to several extra lines of code necessary. Consider the following examples.

The first example turns on the red LED without using the Circuit Playground library.

```
import digitalio
import board

led = digitalio.DigitalInOut(board.D13)
led.direction = digitalio.Direction.OUTPUT

led.value = True
```

The second example turns on the red LED using the Circuit Playground library.

```
from adafruit_circuitplayground import cp

cp.red_led = True
```

Instead of including the setup in the program, the Circuit Playground library includes all the setup in `__init__()` within the module, so setup is automatically done when you import the library. This is a simple example for comparison; some hardware requires significantly more setup than the red LED.

The Library on Circuit Playground Express

The Circuit Playground library has always pushed the memory limits on the Circuit Playground Express. This led us to include the Circuit Playground library and all of its dependencies in the CircuitPython build for Circuit Playground Express as frozen modules.

Frozen modules are library modules that are "frozen" into, or built into, CircuitPython. Freezing a module into CircuitPython moves execution to the flash to save RAM. Normally when a module is imported, the following occurs:

- If it is a .py file, it is compiled into byte codes, which are put in RAM and executed there.
- If it is a .mpy file, it is already compiled into byte codes, and they are put in RAM.

Both of these options use available RAM. This module is complex enough that it quickly outgrew the available RAM on the Circuit Playground Express. So, instead, we freeze the .mpy file into CircuitPython. A frozen .mpy file is already compiled into byte codes like any .mpy file, but the byte codes are already in directly accessible memory (flash), so they don't have to be copied in RAM. This saves on RAM.

In short, it allows us to run a module that would normally run out of memory on import and cause a memory allocation failure. It also means that to use the library with Circuit Playground Express, you simply need to install CircuitPython as the library and all of its dependencies are included in the build.

Normally, you load library modules onto your microcontroller board and place them in the `lib` folder. However, as explained, this module will not function if it is running from the local copy. The `express` module uses `sys.path` on import to specify where the library module should be pulled from. It prefers frozen modules over those contained within the `/lib` folder to ensure that if a user installs the library locally in the `/lib` folder, it will still use the frozen module. It also, however, check the root directory first. In order, it checks root, then frozen, then the `/lib` folder. This order was put in place to ensure the ability to test libraries locally without modifying the library. If you wish to test modifications to one of the modules frozen into CircuitPython for Circuit Playground Express, place the library file in your root directory and it will use that version.

Memory Allocation Failure on Circuit Playground Express

You may find with larger amounts of code or more complicated projects involving external peripherals, that your code fails to run and returns a `MemoryError` in the serial console. The Circuit Playground library includes all the imports and setup necessary to use all of the functionality it provides. This means that it has a relatively large memory footprint. The Circuit Playground Express has limited memory available. The library was designed to make it easy to get started with Circuit Playground and all of the fun stuff that is built in. If you try to use it with a significant amount of code or with many other libraries, you'll find that you may run out of memory on your board. If this happens from either of these scenarios, you're probably ready to move on to using the individual libraries necessary for the hardware you're trying to utilise in your code. This means you would use basic CircuitPython to "manually" initialise all of the hardware you intend to use, instead of relying on the Circuit Playground library. This allows you to initialise only the hardware you will use in your project versus the Circuit Playground library initialising all available features of the CPX.