adafruit learning system

Sensor Plotting with Mu and CircuitPython

Created by Kattni Rembor

Last updated on 2018-09-06 08:05:52 PM UTC

Guide Contents

Light

We're going to use CircuitPython, Mu, and the light sensor on Circuit Playground Express to plot light levels. We have a simple nine-line piece of code below. We'll run this code on our Circuit Playground Express and use Mu to plot the sensor data that CircuitPython prints out.

https://adafru.it/ANO

https://adafru.it/ANO

Save the following as **code.py** on your Circuit Playground Express board, using the Mu editor:

```
import time

import analogio
import board

light = analogio.AnalogIn(board.LIGHT)

while True:
    print((light.value,))
    time.sleep(0.1)
```

On the first few lines we import `analogio`, `board` and `time`, and setup our light sensor. Inside our `while` loop, we are `print()` -ing the light level. The `time.sleep()` is to keep from spamming the results - if they're too fast, they get difficult to read!

Note that the Mu plotter looks for **tuple** values to print. 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((light.value,))`.

Once you have everything setup and running, try placing your hand over the Circuit Playground Express, and watch the plotter immediately react! When you block the light from reaching the CPX, the graphing plotter line goes down. If you shine a flashlight on it to add more light, the plotter goes up!

It's a really easy way to test out your sensors and get a feeling for analog reads in CircuitPython!

Note you can have the text REPL on the left and resize the plotter to be big and on the right like above. That way you see the numbers and the graph at the same time. The plotter will **automatically re-scale** depending on light levels.

Temperature

We're going to use CircuitPython, Mu and the temperature sensor built into the Circuit Playground Express to plot temperature change. We'll run this code on our Circuit Playground Express and use Mu to plot the sensor data that CircuitPython prints out.

https://adafru.it/ANO

https://adafru.it/ANO

Save the following as **code.py** on your Circuit Playground Express board, using the Mu editor:

```
import time

import adafruit_thermistor
import board

thermistor = adafruit_thermistor.Thermistor(
    board.TEMPERATURE, 10000, 10000, 25, 3950)

while True:
    print((thermistor.temperature,))
    # print(((thermistor.temperature * 9 / 5 + 32),))  # Fahrenheit
    time.sleep(0.25)
```

Our code is quite simple. We import `adafruit_thermistor`, `board` and `time`. Then we setup our temperature sensor. Inside our `while` loop, we `print` the temperature in Celsius.

If you'd like to see Fahrenheit instead, place a `#` (# + space) before the line `print((thermistor.temperature,))` and remove the `#` before the line `print(((thermistor.temperature * 9 / 5 + 32),))`. The temperature is in Celsius by default, so we include a little math (the `* 9 / 5 + 32`) to convert it to Fahrenheit.

Note that the Mu plotter looks for **tuple** values to print. 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((thermistor.temperature,))`.

Once you have everything loaded and running, you can place your finger over the temperature sensor to see the plotter immediately respond! Try breathing on your CPX temperature sensor to increase the temperature and watch the plotter go up! Try setting it on a cool surface to watch the plotter go down!

This is a great way to tell the temperature and plot temperature changes in Celsius (or Fahrenheit).


Mu 1.0.0-beta.15 - code.py
code.py
1 import adafruit_thermistor
2 import board
3 import time
4
5 thermistor = adafruit_thermistor.Thermistor(board.TEMPERATURE, 10000, 10000, 25, 3950)
6
7 while True:
8 print((thermistor.temperature,))
9 # print(((thermistor.temperature * 9 / 5 + 32),))
10 time.sleep(0.25)
(20.8015,)
(20.4349,)
(20.1548,)
(20.0039,)
(19.9609,)
(21.1686,)
(21.9469,)
(22.4237,)
(22.619,)
(22.771,)
(22.8362,)
(22.7493,)
(22.5322,)
(22.4237,)
(22.2069,)
31
15
0
-15
-31
Adafruit

Buttons and Switch

We're going to use CircuitPython, Mu, and the two buttons and one switch on Circuit Playground Express to plot button presses and the switch location. We'll run this code on our Circuit Playground Express and use Mu to plot the data that CircuitPython prints out.

https://adafru.it/ANO

https://adafru.it/ANO

Save the following as **code.py** on your Circuit Playground Express board, using the Mu editor:

```
import time

import board
import digitalio

button_a = digitalio.DigitalInOut(board.BUTTON_A)
button_a.direction = digitalio.Direction.INPUT
button_a.pull = digitalio.Pull.DOWN

button_b = digitalio.DigitalInOut(board.BUTTON_B)
button_b.direction = digitalio.Direction.INPUT
button_b.pull = digitalio.Pull.DOWN

switch = digitalio.DigitalInOut(board.SLIDE_SWITCH)
switch.direction = digitalio.Direction.INPUT
switch.pull = digitalio.Pull.UP

while True:
    if button_a.value:
        print((1,))
    elif button_b.value:
        print((2,))
    elif switch.value:
        print((-1,))
    else:
        print((0,))
    time.sleep(0.1)
```

Our code is very simple. First we import `digitalio`, `board` and `time`. Then, we create the `button_a`, `button_b` and `switch` objects.

Our main loop consists of `if`, `elif`, and `else` statements. An `elif` statement says "otherwise, if" something happens, do a thing. An `else` statement says "otherwise do" a thing regardless of what is happening. So, our main loop reads: if button A is pressed, `print` 1, otherwise, if button B is pressed, `print` 2, otherwise, if the switch is moved to the right, `print` -1, otherwise `print` 0. Therefore, it will `print` 0 until you press a button or move the switch. Then we have a `time.sleep(0.1)` to keep from spamming the results - if they move too quickly they're difficult to read!

Note that the Mu plotter looks for **tuple** values to print. 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((1,))`, `print((2,))`, etc.

Once you have everything setup and running, try pressing one of the buttons on the Circuit Playground Express, and watch the plotter immediately react! When you press button A or button B, the plotter line goes up. If you move the switch to the right, the plotter line goes up!

This is a great way to test the buttons and switch on the Circuit Playground Express and plot different numbers simply by printing them!

Motion

We're going to use CircuitPython, Mu and the accelerometer built into the Circuit Playground Express to plot motion. We'll run this code on our Circuit Playground Express and use Mu to plot the motion data that CircuitPython prints out.

https://adafru.it/ANO

https://adafru.it/ANO

Save the following as **code.py** on your Circuit Playground Express board, using the Mu editor:

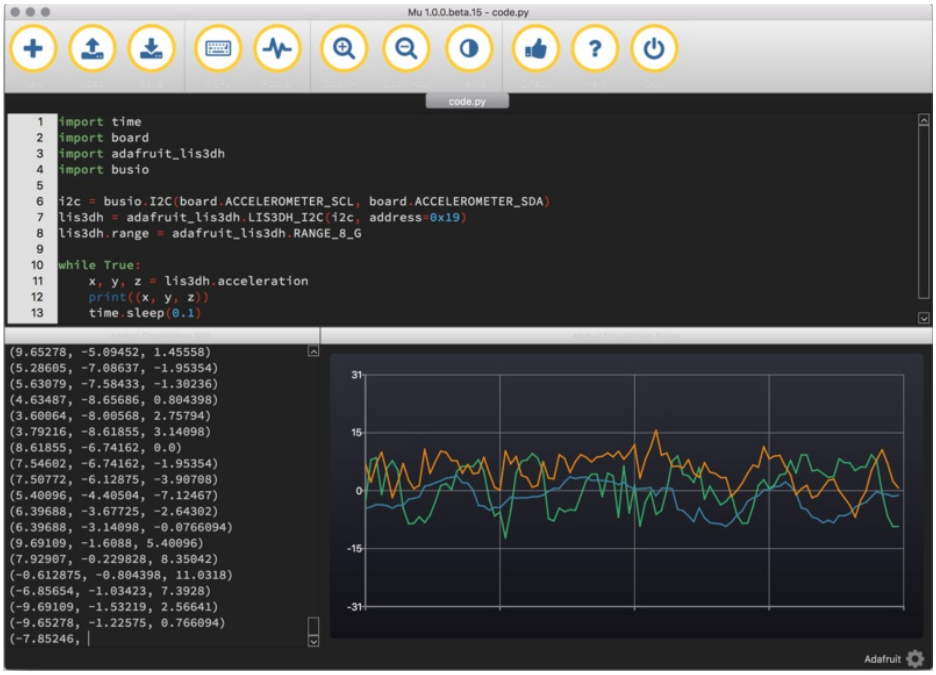
```
import time
import board
import adafruit_lis3dh
import busio

i2c = busio.I2C(board.ACCELEROMETER_SCL, board.ACCELEROMETER_SDA)
lis3dh = adafruit_lis3dh.LIS3DH_I2C(i2c, address=0x19)
lis3dh.range = adafruit_lis3dh.RANGE_8_G

while True:
    x, y, z = lis3dh.acceleration
    print((x, y, z))
    time.sleep(0.1)
```

Our code is really simple. We import `time`, `board`, `adafruit_lis3dh`, and `busio`. Then we setup the accelerometer. Inside our while loop, we assign `x, y, z` to be the motion values. Then we `print` our motion values. The `time.sleep(0.1)` slows down the readings a little - without it they're too fast to read!

Note that the Mu plotter looks for **tuple** values to print. Tuples in Python come in parentheses () with comma separators. If you have two values, a tuple would look like `(1.0, 3.14)`. We have three values, `(x, y, z)`, and note that the tuple itself has its own parentheses. Thus the extra parentheses in `print((x, y, z))`.

Once you have everything up and running, try moving the board around to see the plotter respond immediately! The accelerometer is in the center of the board, and depending on which axis you move the board around, the x, y or z values will change. Try moving it only one direction to watch a single value change! Try shaking it all around to see all the values change significantly!

This is a great way to see the motion values and plot the motion as you move the board!


Mu 1.0.0.beta.15 - code.py
code.py
1 import time
2 import board
3 import adafruit_lis3dh
4 import busio
5
6 i2c = busio.I2C(board.ACCELEROMETER_SCL, board.ACCELEROMETER_SDA)
7 lis3dh = adafruit_lis3dh.LIS3DH_I2C(i2c, address=0x19)
8 lis3dh.range = adafruit_lis3dh.RANGE_8_G
9
10 while True:
11 x, y, z = lis3dh.acceleration
12 print((x, y, z))
13 time.sleep(0.1)
(9.65278, -5.09452, 1.45558)
(5.28605, -7.08637, -1.95354)
(5.63079, -7.58433, -1.30236)
(4.63487, -8.65686, 0.804398)
(3.60064, -8.00568, 2.75794)
(3.79216, -8.61855, 3.14098)
(8.61855, -6.74162, 0.0)
(7.54602, -6.74162, -1.95354)
(7.50772, -6.12875, -3.90708)
(5.40096, -4.40504, -7.12467)
(6.39688, -3.67725, -2.64302)
(6.39688, -3.14098, -0.0766094)
(9.69109, -1.6088, 5.40096)
(7.92907, -0.229828, 8.35042)
(-0.612875, -0.804398, 11.0318)
(-6.85654, -1.03423, 7.3928)
(-9.69109, -1.53219, 2.56641)
(-9.65278, -1.22575, 0.766094)
(-7.85246,
31
15
0
-15
-31
Adafruit

Sound

We're going to use CircuitPython, Mu and the sound sensor built into the Circuit Playground Express to plot sound levels. We'll run this code on our Circuit Playground Express and use Mu to plot the sensor data that CircuitPython prints out.

https://adafru.it/ANO

https://adafru.it/ANO

Save the following as **code.py** on your Circuit Playground Express board, using the Mu editor:

```
import array
import math
import time

import audiobusio
import board


def mean(values):
    return sum(values) / len(values)


def normalized_rms(values):
    minbuf = int(mean(values))
    sum_of_samples = sum(
        float(sample - minbuf) * (sample - minbuf)
        for sample in values
    )

    return math.sqrt(sum_of_samples / len(values))


mic = audiobusio.PDMIn(
    board.MICROPHONE_CLOCK,
    board.MICROPHONE_DATA,
    frequency=16000,
    bit_depth=16
)
samples = array.array('H', [0] * 160)
mic.record(samples, len(samples))

while True:
    mic.record(samples, len(samples))
    magnitude = normalized_rms(samples)
    print(((magnitude),))
    time.sleep(0.1)
```

Let's look at the code!

First we import `audiobusio`, `time`, `board`, `array` and `math`. Then we have two helper functions. The first one uses math to return a mean, or average. It is used in the second helper. The second one uses math to return a normalised rms average (https://adafru.it/Bf5). We use these functions to take multiple sound samples really quickly and average them to get a more accurate reading.

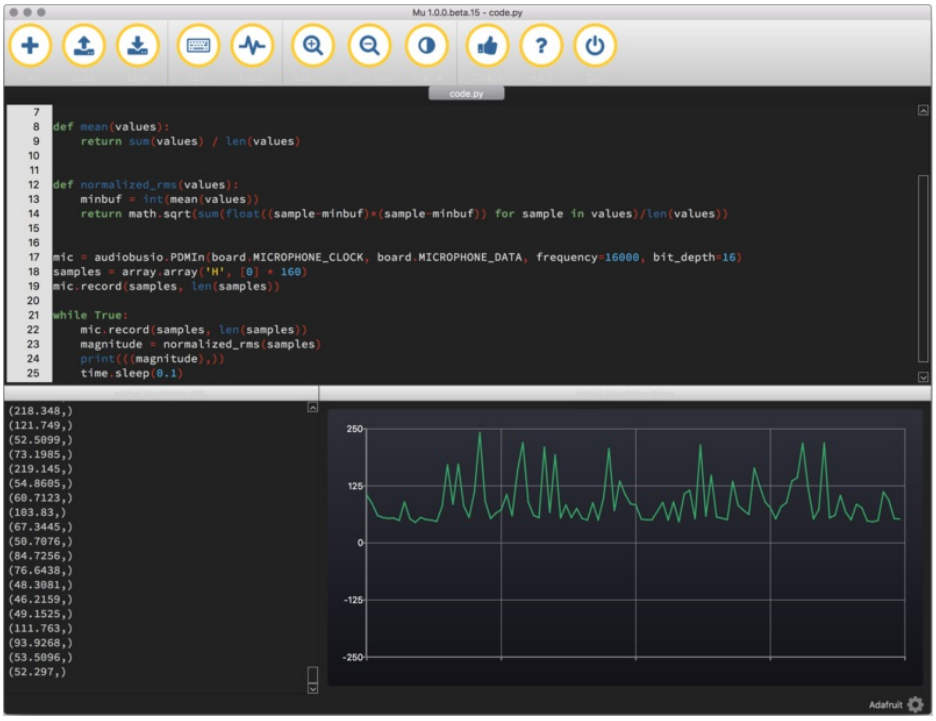
Next we set up the microphone object and our samples variable. Then we initialise the mic object so it's ready when we are.

Then we use the mic object to start taking sound samples. We use our normalised rms to find the average of a given set of samples, and we call that the `magnitude`. Last, we `print` the `magnitude` to the serial console.

Note that the Mu plotter looks for **tuple** values to print. 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(((magnitude),))`.

Once you have everything setup and running, try speaking towards the Circuit Playground Express, and watch the plotter immediately react! Move further away from the board to cause smaller changes in the plotter line. Move closer to the board to see bigger spikes!

It's a really easy way to test your microphone and see how it reads sound changes on the Circuit Playground Express!



Capacitive Touch

We're going to use CircuitPython, Mu and the capacitive touch pads built into the Circuit Playground Express to plot the raw capacitive touch values. We'll run this code on our Circuit Playground Express and use Mu to plot the capacitive touch data that CircuitPython prints out.

https://adafru.it/ANO

https://adafru.it/ANO

Save the following as **code.py** on your Circuit Playground Express board, using the Mu editor:

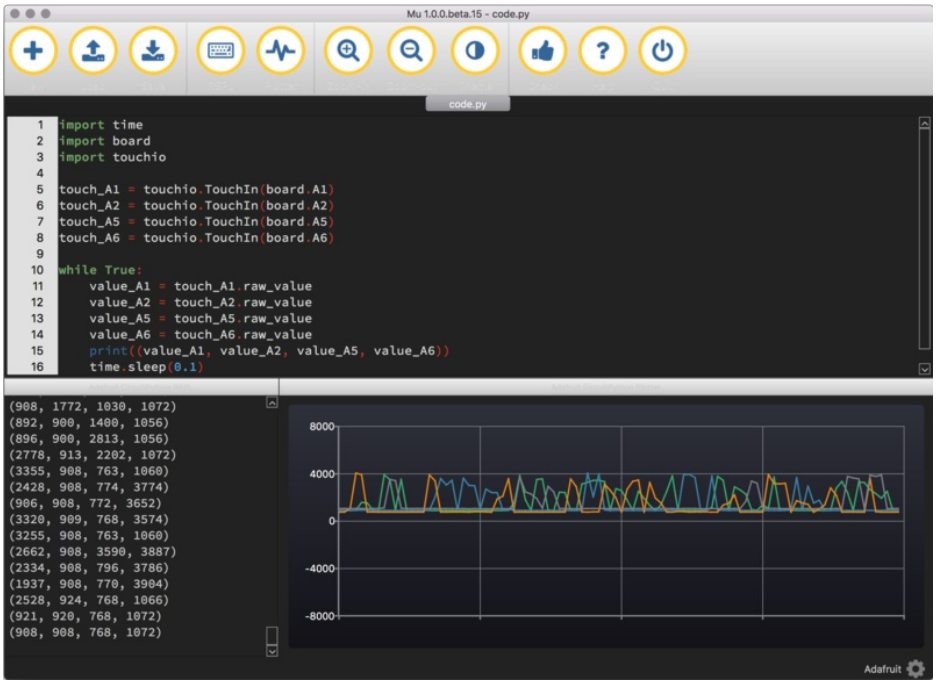
```
import time

import board
import touchio

touch_A1 = touchio.TouchIn(board.A1)
touch_A2 = touchio.TouchIn(board.A2)
touch_A5 = touchio.TouchIn(board.A5)
touch_A6 = touchio.TouchIn(board.A6)

while True:
    value_A1 = touch_A1.raw_value
    value_A2 = touch_A2.raw_value
    value_A5 = touch_A5.raw_value
    value_A6 = touch_A6.raw_value
    print((value_A1, value_A2, value_A5, value_A6))
    time.sleep(0.1)
```

Our code is fairly simple. First we import `time`, `touchio` and `board`. Next, we setup four touch pads, A1, A2, A5 and A6. Inside our `while` loop, we assign `value_PadName` to the `raw_value` for each pad, e.g. `value_A1 = touch_A1.raw_value`, etc. Then we `print` those values in a tuple. And last, we have a `time.sleep(0.1)` to slow down the reading of the values - if it's too fast, it's really hard to read!

Note that the Mu plotter looks for **tuple** values to print. Tuples in Python come in parentheses () with comma separators. If you have two values, a tuple would look like `(1.0, 3.14)`. We have four values, `(value_A1, value_A2, value_A5, value_A6)`, and note that the tuple itself has its own parentheses. Thus the extra parentheses in `print((value_A1, value_A2, value_A5, value_A6))`.

Once you have everything up and running, try touching any of the four pads, A1, A2, A5 or A6 to see the plotter respond immediately! Try touching only one pad to see of the four lines in the plotter go up. Try touching more than one pad at once to see multiple lines move. The harder you touch the pad, the higher the plotter line will go!

This is a great way to test your capacitive touch pads and plot the changes as you touch different pads!


Mu 1.0.0.beta.15 - code.py
code.py
import time
import board
import touchio
touch_A1 = touchio.TouchIn(board.A1)
touch_A2 = touchio.TouchIn(board.A2)
touch_A5 = touchio.TouchIn(board.A5)
touch_A6 = touchio.TouchIn(board.A6)
while True:
value_A1 = touch_A1.raw_value
value_A2 = touch_A2.raw_value
value_A5 = touch_A5.raw_value
value_A6 = touch_A6.raw_value
print((value_A1, value_A2, value_A5, value_A6))
time.sleep(0.1)
(908, 1772, 1030, 1072)
(892, 900, 1400, 1056)
(896, 900, 2813, 1056)
(2778, 913, 2202, 1072)
(3355, 908, 763, 1060)
(2428, 908, 774, 3774)
(906, 908, 772, 3652)
(3320, 909, 768, 3574)
(3255, 908, 763, 1060)
(2662, 908, 3590, 3887)
(2334, 908, 796, 3786)
(1937, 908, 770, 3904)
(2528, 924, 768, 1066)
(921, 920, 768, 1072)
(908, 908, 768, 1072)
8000
4000
0
-4000
-8000
Adafruit

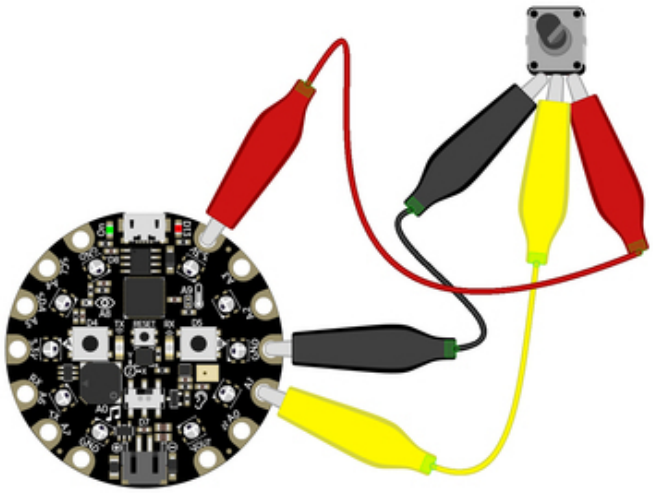
Potentiometer

We're going to use CircuitPython, Mu, and a potentiometer with Circuit Playground Express to plot voltage levels. We'll run this code on our Circuit Playground Express and use Mu to plot the voltage data that CircuitPython prints out.

https://adafru.it/ANO

https://adafru.it/ANO

First, let's get the potentiometer attached to your Circuit Playground Express!



To connect the potentiometer to the Circuit Playground Express:

- Connect the **left pin** on the potentiometer to **GND** on the Circuit Playground Express.
- Connect the **middle pin** on the potentiometer to **A1** on the Circuit Playground Express.
- Connect the **right pin** on the potentiometer to **3.3V** on the Circuit Playground Express.

Save the following as **code.py** on your Circuit Playground Express board, using the Mu editor:

```
import time

import analogio
import board

potentiometer = analogio.AnalogIn(board.A1)


def get_voltage(pin):
    return (pin.value * 3.3) / 65536


while True:
    print((get_voltage(potentiometer),))
    time.sleep(0.1)
```

Let's take a look at the code!

First we import `time`, `analogio` and `board`. Next we create the `potentiometer` object and assign it to pin `A1`.

Then we have the `get_voltage()` helper function. By default, analog readings will range from 0 (minimum) to 65535 (maximum). This helper will convert the 0-65535 reading from `pin.value` and convert it a 0-3.3V voltage reading.

Our main loop is super simple. Inside our `print` statement, we call the `get_voltage()` helper and provide it with the `potentiometer` object. Then we have a `time.sleep(0.1)` to slow down the printed results - otherwise they'd be too fast to read!

Note that the Mu plotter looks for **tuple** values to print. 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((get_voltage(potentiometer),))` .

Once you have everything setup and running, try rotating the potentiometer knob attached the Circuit Playground Express, and watch the plotter immediately react! Rotate to the left to watch the plotter go down. Rotate to the right to watch it go up!

This is a great way to see the voltage changes resulting from using a potentiometer, and plot the changes as you move the knob!

Color

We're going to use CircuitPython, Mu, and the light sensor on Circuit Playground Express to plot color levels. We'll run this code on our Circuit Playground Express and use Mu to plot the color data that CircuitPython prints out.

https://adafru.it/ANO

https://adafru.it/ANO

Save the following as **code.py** on your Circuit Playground Express board, using the Mu editor:

```
import analogio
import board
import neopixel

pixels = neopixel.NeoPixel(board.NEOPIXEL, 10, brightness=1.0)
light = analogio.AnalogIn(board.LIGHT)

while True:
    pixels.fill((0, 0, 0))
    pixels[1] = (255, 0, 0)
    raw_red = light.value

    red = int(raw_red * (255 / 65535))
    pixels[1] = (0, 255, 0)
    raw_green = light.value

    green = int(raw_green * (255 / 65535))
    pixels[1] = (0, 0, 255)
    raw_blue = light.value

    blue = int(raw_blue * (255 / 65535))
    pixels.fill((0, 0, 0))

    # Printed to match the color lines on the Mu plotter!
    # The orange line represents red.
    print((green, blue, red))
```

Let's take a look at the code!

First we import `neopixel`, `analogio`, `time` and `board`. Next, we create the `pixels` object for the NeoPixels and the `light` object for the light sensor.

LED colors are set using a combination of **r**ed, **g**reen, and **b**lue, in the form of an (**R**, **G**, **B**) tuple. Each member of the 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.

The light sensor works as a color sensor by using the NeoPixel next to it (pixel 1) to flash red, green and blue, and then record the reflected light levels of each color to determine the color of the object being held against it. So, inside our main loop, we need to flash red, green and blue, record the reflected light levels, and then use math to determine the level of each color.

The first thing we do in our main loop is make sure the NeoPixels are off with `pixels.fill((0, 0, 0))`. Next, we begin with red. We flash `pixel[1]` red by assigning it `(255, 0, 0)`, for `0.5` seconds. Then we grab the light sensor value and assign it to `raw_red`.

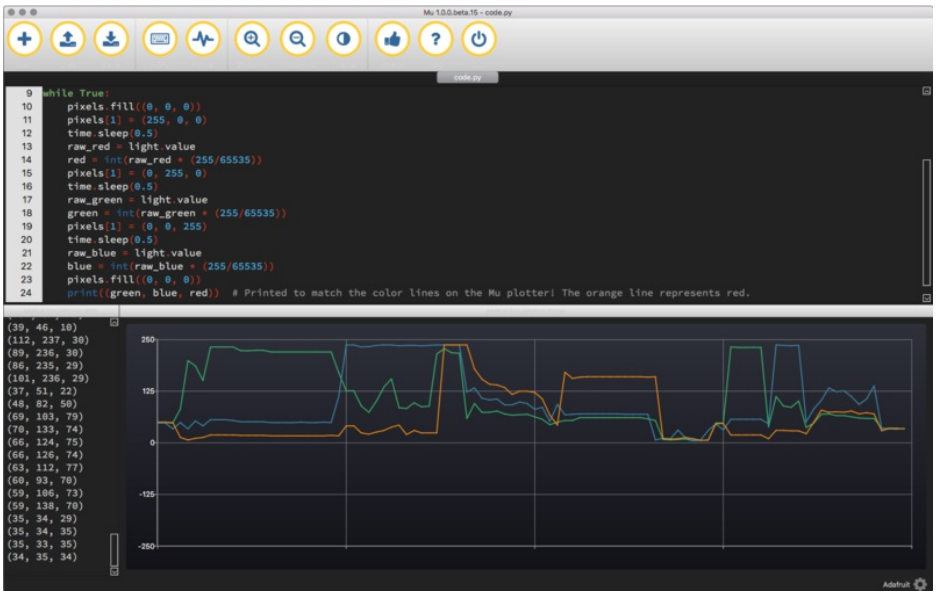
The light sensor returns a value between 0 and 65535. Since color values are 0-255, we need to use math to scale the raw light sensor value down to a viable color value. So, to get our red value, we use `red = int(raw_red * (255/65535))`. This takes whatever value the light sensor provides, and returns an equivalent value between 0 and 255. Now we have our red level!

We repeat the exact same steps for green, and then for blue.

The last thing we do, is `print` our `red`, `green` and `blue` values in the form of a tuple, so we can see them on the plotter! We've changed the order a bit to `print((green, blue, red))` so the color of the lines on the plotter match the colors they represent: green for green, blue for blue and orange for red.

Once you have everything setup and running, try holding a colored item up to the light sensor on the Circuit Playground Express, and watch the plotter immediately react! Hold up a green item to watch the green line go higher than the blue or red. Hold up a red item and watch the orange line go higher than the green or the blue!

This is a great way to see color levels sense using the light sensor, and plot the changes as you hold up different colors!



Heartbeat Pulse

We're going to use CircuitPython, Mu, and the light sensor on Circuit Playground Express to plot pulse sensing. We'll run this code on our Circuit Playground Express and use Mu to plot the pulse data that CircuitPython prints out.

https://adafru.it/ANO

https://adafru.it/ANO

Save the following as **code.py** on your Circuit Playground Express board, using the Mu editor:

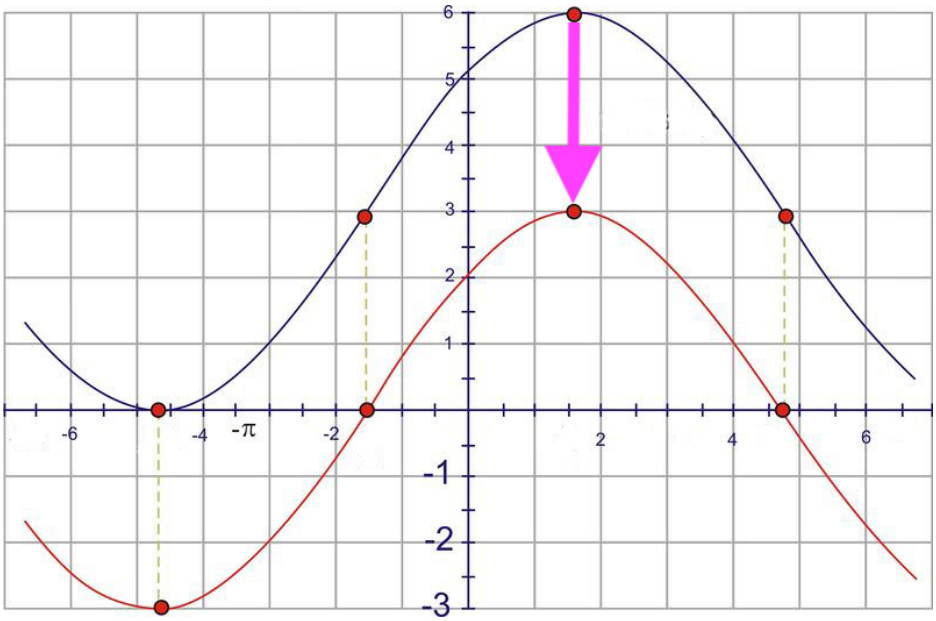
```
import time

import analogio
import board
import neopixel

pixels = neopixel.NeoPixel(board.NEOPIXEL, 10, brightness=1.0)
light = analogio.AnalogIn(board.LIGHT)

# Turn only pixel #1 green
pixels[1] = (0, 255, 0)

# How many light readings per sample
NUM_OVERSAMPLE = 10
# How many samples we take to calculate 'average'
NUM_SAMPLES = 20
samples = [0] * NUM_SAMPLES

while True:
    for i in range(NUM_SAMPLES):
        # Take NUM_OVERSAMPLE number of readings really fast
        oversample = 0
        for s in range(NUM_OVERSAMPLE):
            oversample += float(light.value)
        # and save the average from the oversamples
        samples[i] = oversample / NUM_OVERSAMPLE  # Find the average

        mean = sum(samples) / float(len(samples))  # take the average
        print((samples[i] - mean,))  # 'center' the reading
        time.sleep(0.025)  # change to go faster/slower
```

For a detailed explanation of how LED pulse sensing works, check out this article (https://adafru.it/BfB).

There are two things we have to do.

First, the values that result from pulse sensing are often noisy or jittery: some are too high, and some are too low, so we smooth them out by taking an average. We take 10 readings as fast as possible (faster than we need to), find the average, and that smooth out the noise. This is called oversampling.

Second, we want to center the readings around zero. The original samples are all the values are positive (greater than or equal to zero). To center the values around zero, we find the average and shift all the values down. So instead of the value always being greater than zero, it will vary above and below zero, with the average being zero. This is called "removing the DC bias" on the signal. To learn more about DC bias, check out this article (https://adafru.it/BfC).



Since the signal keeps changing, the average is also going to keep changing. So we keep the last 20 samples and compute their average. When the next sample comes along, we drop the oldest sample, and recompute the average again of the 20 most recent values. This is a called a "moving average". Picture a moving window that is 20 samples wide, moving along the stream of sample data that we are taking. For more detailed information about moving averages, check out this article (https://adafru.it/BfD).

We begin our code by importing the modules we need: `neopixel`, `analogio`, `time` and `board`. Next, we create the `pixels` object for the NeoPixels and the `light` object for the light sensor.

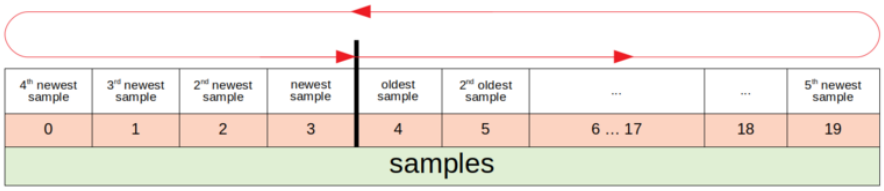
Then, we turn the pixel next to the light sensor green. Note that we turned pixel number "1" green, but it's actually the second pixel on the board. This is because Python starts counting with 0, so the first pixel is pixel number "0".

Next, we assign `NUM_OVERSAMPLE = 10` to specify how many light readings we'll take per sample, and `NUM_SAMPLES = 20` to specify how many samples we're going to take to calculate the average we need to remove the DC bias. Then we create a place to store the samples for the moving average, in `samples = [0] * NUMSAMPLES`.

Now we start reading samples. The outer loop `for i in range(NUM_SAMPLES):` tells the code to cycle through a range of 20 values, so `i` keeps running between 0 and 19 continuously.

As we mentioned above, we are also going to take 10 samples as a fast as possible and average them to smooth out the noise. The inner loop `for s in range(NUM_OVERSAMPLE):` reads 10 values from the light sensor. We sum up all those oversampling values using `oversample += float(light.value)`, until we've added up `NUM_SAMPLES` number of values. Then we divide by 10 to find the average of the oversampling values, and store the value with `samples[i] = oversample / NUM_OVERSAMPLE`.

After we've computed the oversampling average, we compute the moving average with `mean = sum(samples) / float(len(samples))`. The window wraps around the `samples` array. For instance, if `i` is 3, the most recent sample is in `samples[3]`, and the previous samples are in `samples[2]`, `samples[1]`, `samples[0]`, and then wraps around back to `samples[19]`, `samples[18]`, etc. all the way back to `samples[4]`, the oldest value. This image provides a visual explanation.



We subtract the average from the sampled value, with `samples[i] - mean,` to center the reading around zero, or remove the DC bias.

Remember, we initialised the moving average samples array with `samples = [0] * NUMSAMPLES`. You'll notice the plotter doesn't respond correctly at the very beginning. This is because the the moving average that we're taking while the first 20 samples are still in our moving window includes the zeros that we started with before our main loop. So, until we've moved past the first 20 samples, our average will be skewed.

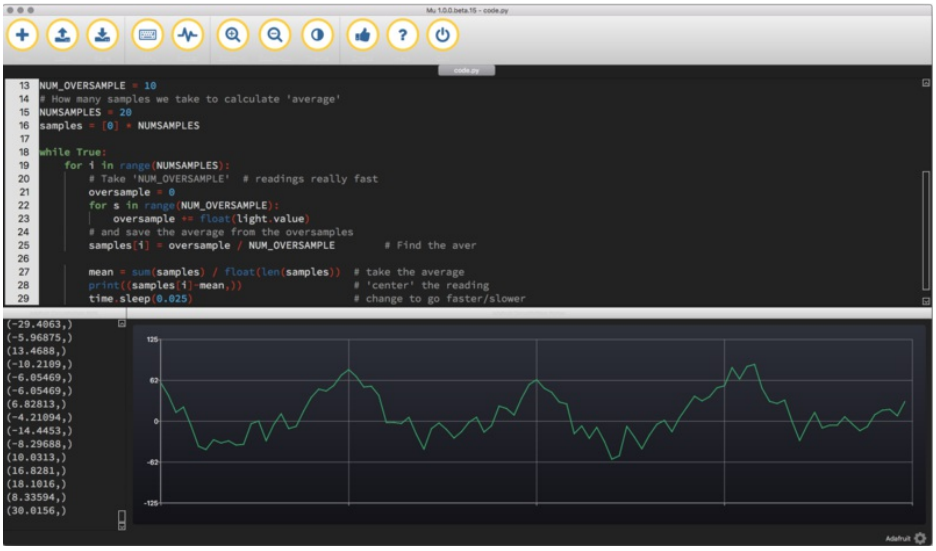
Then, we `print` it as a tuple `print((samples[i] - mean,))`.

Note that the Mu plotter looks for **tuple** values to print. 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((samples[i] - mean,))`.

Finally we include `time.sleep(0.025)` to give a slight delay to the readings.

Once you have everything setup and running, try pressing your finger over the green LED and the light sensor on the Circuit Playground Express, and watch the plotter react! If you press too hard, sometimes it won't respond. But if you press lightly, you'll see a wave form on the plot that matches your pulse!

This is a great way to sense your pulse using the light sensor, and watch plot the changes as you press your finger against it!



Soil Moisture

We're going to use CircuitPython, Mu, and the capacitive touch pads on Circuit Playground Express to plot soil moisture sensing. We'll run this code on our Circuit Playground Express and use Mu to plot the soil moisure data that CircuitPython prints out.

https://adafru.it/ANO

https://adafru.it/ANO

The hardware you'll need for this project is the Circuit Playground Express, an alligator clip and a nail. You'll also need at least two soil samples, one wet and one dry, for calibrating the code.

Assembly is as simple as connecting one end of your alligator clip to pad A1 on the Circuit Playground Express, and the other end of your alligator clip to the top of your nail.

Save the following as **code.py** on your Circuit Playground Express board, using the Mu editor:

```
import time
from adafruit_circuitplayground.express import cpx
import touchio
import simpleio
import board

cpx.pixels.brightness = 0.2
touch = touchio.TouchIn(board.A1)

DRY_VALUE = 1500  # calibrate this by hand!
WET_VALUE = 2100  # calibrate this by hand!

while True:
    value_A1 = touch.raw_value
    print((value_A1,))

    # fill the pixels from red to green based on soil moisture
    percent_wet = int(simpleio.map_range(value_A1, DRY_VALUE, WET_VALUE, 0, 100))
    cpx.pixels.fill((100-percent_wet, percent_wet, 0))
    time.sleep(0.5)
```

First we import the libraries we need. The first library is `time`. Next, you'll see we import a specific part of of the second library by saying `from adafruit_circuitplayground.express import cpx`. This way we can call this library by typing `cpx` instead of the longer library name. Then we import `touchio`, `simpleio`, and `board`.

We set the pixel brightness. Pixel brightness is set by a number between 0 and 1, representing 0-100%, i.e. 0.3 would be 30%. So we set our brightness to 20% with `cpx.pixels.brightness = 0.2`. Then we create the `touch` object so we can use it as our moisture sensor. We are going to use pin A1 in this project, so we provide `board.A1`.

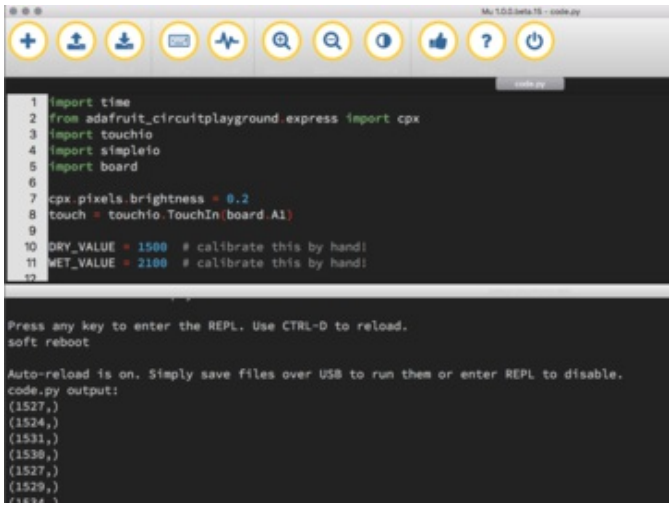
To sense whether the soil is dry or wet, we're going to use the raw capacitive touch values from the capacitive touch pad on the Circuit Playground Express. We create two variables, `DRY_VALUE` and `WET_VALUE` and assign them to the raw capacitive touch values associated with dry and wet soil.

We find these values in the main loop. We assign `value_A1 = touch.raw_value`, and then `print` the results of `value_A1`. This allows us to view the results of the raw capactitive touch values in the serial output. Wet soil has a higher raw

capacitive touch value than dry soil, and we can use this to track whether the soil is wet or dry. The code comes with default values for both of these variables, however, the your actual results may be different. Therefore, you should calibrate these values first.

Note that the Mu plotter looks for **tuple** values to print. 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((value_A1,))`.

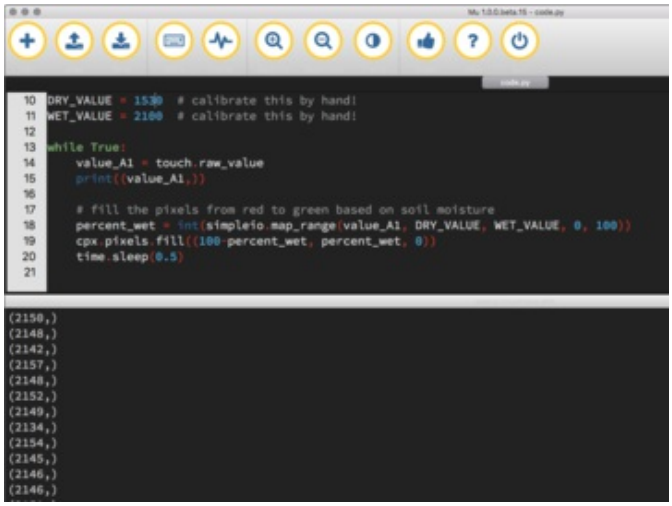
`DRY_VALUE` and `WET_VALUE` both need to be calibrated by hand. To calibrate them, place the nail into your soil sample, and open the serial console (REPL) in Mu. Then you can see what the returned value is and you can change your variables to reflect it.



First place your nail into your dry soil sample.

As you can see, the dry value we are getting back is a little bit different than the default number. So let's change `DRY_VALUE` to reflect our results.

Now we'll do the same to get our wet value.



Place the nail into your wet soil sample.

Now look at the values in the serial output. Our wet value is a little bit different, so let's change `WET_VALUE` as well

We've determined our wet and dry values. Now we'll use these values to make the LEDs green when the soil is 100% wet, and red when the soil is dry, or 0% wet. This is how you'll know your plant needs to be watered!

LED colors are set using a combination of **r**ed, **g**reen, and **b**lue, in the form of an (**R**, **G**, **B**) tuple. Each member of the 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.

So, we're going to need to map the raw capacitive touch values to fit within the values needed for an RGB tuple. To do this, we're going to use `simpleio.map_range`. This works by providing the value we're going to use (`value_A1`), the minimum value we expect value_A1 to be (which is `DRY_VALUE`), the maximum value we expect `value_A1` to be (`WET_VALUE`), and then the minimum and maximum numbers we'd like to match it to, which is `0` and `100`. By assigning `percent_wet = int(simpleio.map_range(value_A1, DRY_VALUE, WET_VALUE, 0, 100))`, we are taking the raw capacitive touch values and mapping them to a whole number (`int`) between 0 and 100 to get a percentage.

We use this percentage to set the LEDs to be red when dry, green when wet, and a yellowish color in between as the soil is slowly drying out. We light up all the pixels with `cpx.pixels.fill((100-percent_wet, percent_wet, 0))`, which uses the `percent_wet` we get from the previous line.

When the soil is 100% wet, the first member of the tuple is `0` (`100-percent_wet` if `percent_wet`=100 is 0), and the second member is `100` (because `percent_wet`=100). So the tuple when the soil is wet is (0, 100, 0) which means the LEDs will be green. This is how you know you don't need to water your soil!

When the soil is 0% wet, the first member of the tuple is `100` (`100-percent_wet` if `percent_wet`=0 is 100), and the second member is `0` (because `percent_wet`=0). So the tuple when the soil is wet is `(100, 0, 0)` which means the LEDs will be red. This is how you know it's time to water your soil!

But what about when your soil is sort of wet and sort of dry? When the soil is 50% wet, the first member of the tuple is `50` (`100-percent_wet` if `percent_wet`=50 is 50), and the second member is `50` (because `percent_wet`=50). So the tuple when the soil is half dry is (50, 50, 0) which means the LEDs will be yellow. The yellow color will range from a greenish-yellow to a reddish-yellow as your soil goes from moist to slowly drying out. This is how you'll know it'll be time to water your soil soon!

Finally, we include a `time.sleep(0.5)` to slow down the speed of reading the data. Soil dries out slowly so there's no need for reading the data super quickly.

Once you have everything setup and running, try placing your nail into wet soil, and watch the plotter immediately react! Place your nail into dry soil to watch the plotter go down. Place it into soil that's only sort of wet to watch it go up a little Place it back into wet soil and watch it go up higher!

This is a great way to see see soil moisture levels using the capacitive touch pad, and plot the changes as the soil slowly dries out!


Mu 1.0.0.beta.15 - code.py
code.py
7 cpx.pixels.brightness = 0.2
8 touch = touchio.TouchIn(board.A1)
9
10 DRY_VALUE = 1500 # calibrate this by hand!
11 WET_VALUE = 2100 # calibrate this by hand!
12
13 while True:
14 value_A1 = touch.raw_value
15 print((value_A1,))
16
17 # fill the pixels from red to green based on soil moisture
18 percent_wet = int(simpleio.map_range(value_A1, DRY_VALUE, WET_VALUE, 0, 100))
19 cpx.pixels.fill((100-percent_wet, percent_wet, 0))
20 time.sleep(0.5)
(1526,)
(1527,)
(1523,)
(1562,)
(1660,)
(1814,)
(1992,)
(1965,)
(2687,)
(2742,)
(2732,)
(2439,)
(2133,)
(2130,)
(2123,)
(2131,)
(2123,)
(2141,)
8000
4000
0
-4000
-8000
Adafruit