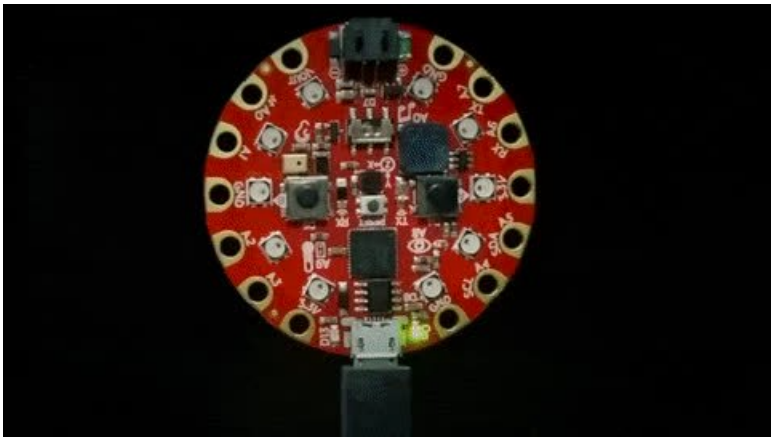
adafruit learning system

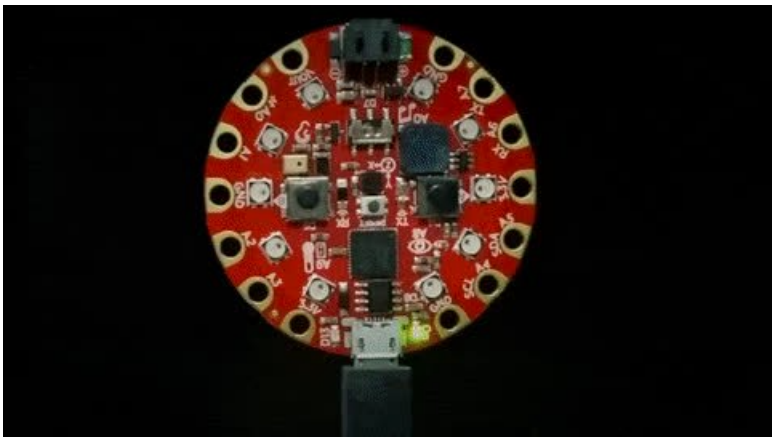
Simon Game Clone with Circuit Playground Express and CircuitPython

Created by Miguel Grinberg



Last updated on 2019-07-24 07:08:34 PM UTC

Overview



In this guide, you will learn how to write a cute and fun little game for the Circuit Playground Express board, using the CircuitPython language. The game is a clone of Simon, a once very popular game of memory in which you have to remember a sequence of colors and sounds that gets progressively longer. This game was an instant hit when it was released in 1978 and became a pop culture symbol of the 1970s and 1980s.

In case you have never heard of Simon before, below you can see the original game console in all its splendor:

How do you play this game? Simon generates a random sequence of light and sound signals that you have to repeat by pressing the buttons in the same order. The first sequence has just one signal so it is very easy, but the game gets progressively harder because each time you complete the sequence, one more signal is added at the end. Take a look at this video in which I play the game as implemented on this guide:

My version of this game for the Circuit Playground Express uses several of the features of this board:

- The ten NeoPixel RGB LEDs are split in groups to represent the four colored areas
- The speaker plays tones that approximate those of the original game console
- The capacitive touch inputs are used for the player to enter the sequence

To write this game, I used a software technique called *bottom-up development*. The basic idea is to take a complex project and break it up into smaller and simpler tasks. You begin by implementing the smaller tasks and then work your way up to the bigger ones, which are now less complex because they can rely on the lower level tasks that are already implemented. This way of tackling a software task is good for many types of projects, but is particularly well suited for games. I hope once you learn it you will be able to apply to your own projects!

Parts

This is an easy, no solder project. All you need is an Adafruit Circuit Playground Express board and a known good USB A to microB cable containing both power and data wires. Do not use the stubby USB cables that come with cell phone charging batteries, most do not have the data wires!

Your browser does not support the video tag.

Circuit Playground Express

$24.95
IN STOCK

ADD TO CART

USB cable - USB A to Micro-B

$2.95
IN STOCK

ADD TO CART

Using the Mu Editor

For this project I'm going to use the Mu Editor, a simple Python code editor tailored to beginners that runs on MacOS, Windows and Linux. If you don't have Mu installed, you can download a copy and learn how to install it on your computer by clicking below:

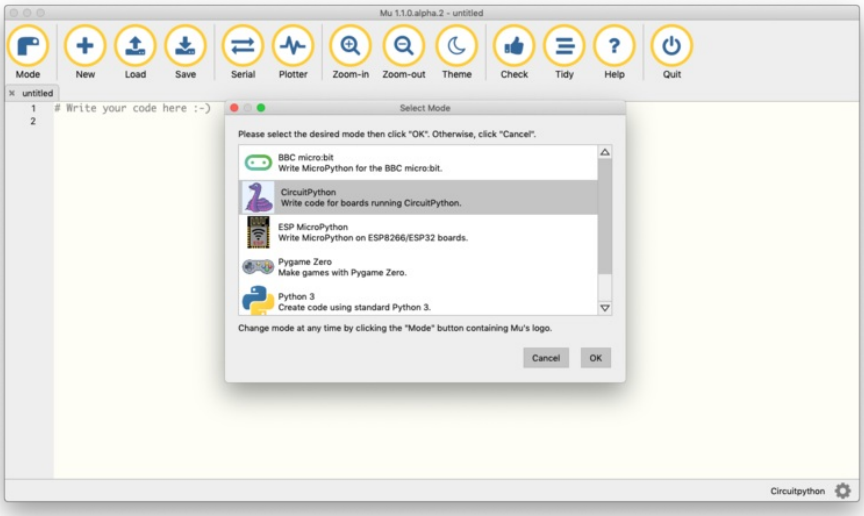
https://adafru.it/Be6

https://adafru.it/Be6

Step by step instructions for Mu are listed here in the Welcome to CircuitPython Guide (https://adafru.it/ANO).

Before you run Mu, connect your board to your computer with a USB cable, so that Mu can recognize it right away when it starts. Note that if you are using a Microsoft Windows 7 or 8 computer, you will need to install the Adafruit drivers for your board if you haven't done that yet. **Drivers are NOT needed for Windows 10**.

Mu uses "modes", a set of predefined configurations tailored to specific uses. The mode is changed by clicking on the first toolbar button on the left, appropriately labeled **Mode**. Depending on the version of Mu that you are using, you are going to find either an "Adafruit" mode, or a "CircuitPython" mode. Make sure this mode is selected.



Click on the **Serial** button and then press **Ctrl-C** (press and hold the Ctrl key, then press the letter C). The Ctrl-C key combination is used to interrupt the code that is currently running on the board. You should now see a message that reads "Press any key to enter the REPL. Use Ctrl-D to reload". Press enter or any other key (except Ctrl-D!), and that should bring you into a Python prompt that looks more or less like this:

```
Press any key to enter the REPL. Use CTRL-D to reload.
Adafruit CircuitPython 4.0.2 on 2019-06-27; Adafruit CircuitPlayground Express with samd21g18
>>>
```

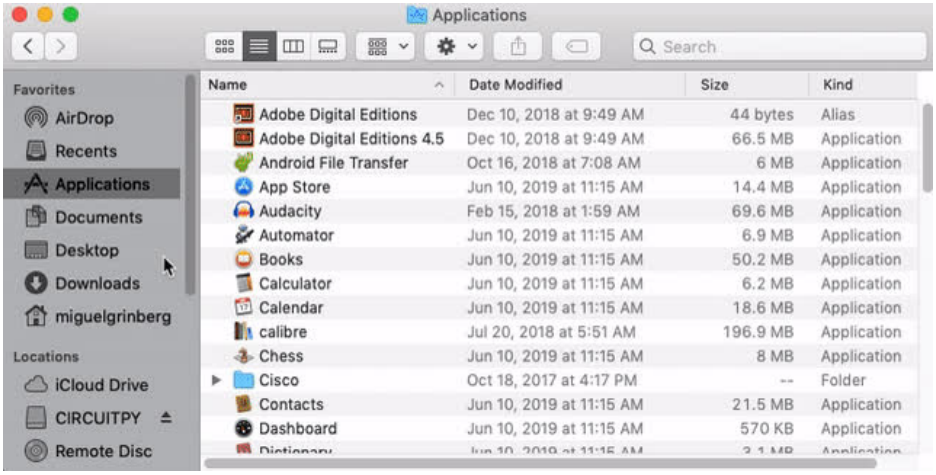
This prompt is called a *REPL*, which is short for Read-Eval-Print-Loop. Here you can type code and have it executed immediately by the CircuitPython interpreter running inside the board. The REPL is an invaluable tool while writing code, since you can use to test things out with immediate feedback. Also, if the code that you write has errors, the

REPL will show you information that can be useful in diagnosing and fixing the problem. I recommend that you keep the REPL window open at all times while working with a CircuitPython board.

Try running a simple Python command in the REPL. For example:

```
>>> print('Hello from CircuitPython!')
Hello from CircuitPython!
>>>
```

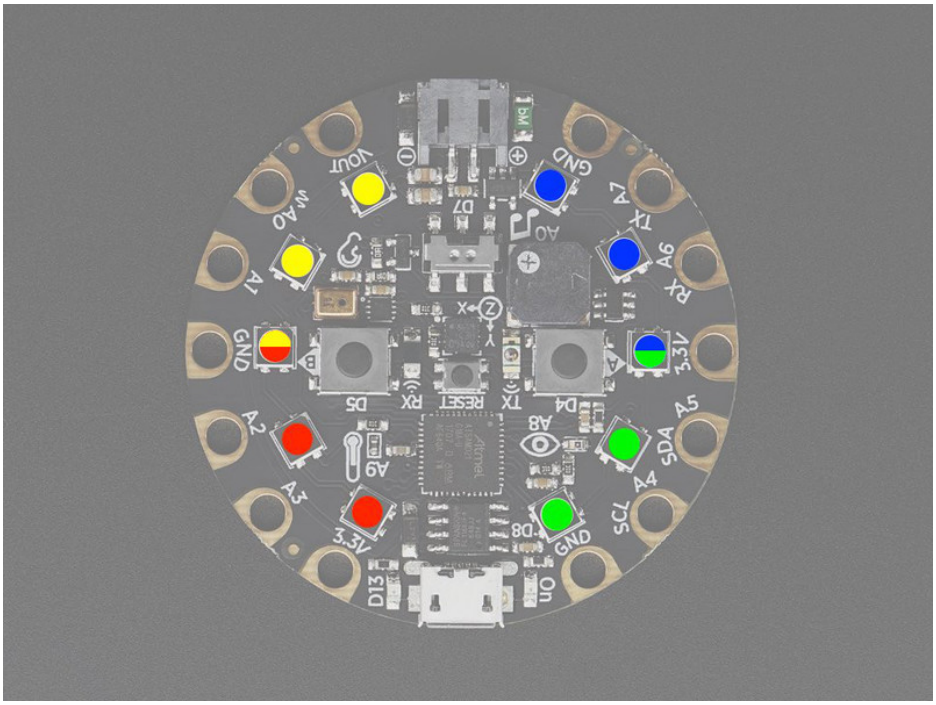
Before I begin with the game, let's spend a moment thinking about the code that is currently stored in your board, which will soon be replaced. I assume you don't want to lose this code, so now is a good time to make a backup copy. In your computer you should have a disk labeled **CIRCUITPY** which represents the board's storage. Inside this disk, you will find a file named **code.py**. You can drag this file into your computer to make a copy of it, or if you prefer you can rename it in place to something such as **code-backup.py**.



Light Design

Okay, let's begin writing the Simon game clone!

Following the bottom-up approach that I described in the introduction, my first goal is going to be to implement the four lights of the original game. The first important decision to make is how the four light regions of the game console are going to be mapped to the ten LEDs in the Circuit Playground Express board. After thinking about it for some time, I decided to use the following layout:



As you can see in the diagram, each of the four light regions is going to use three of the NeoPixel LEDs. Since there are only ten LEDs, the two LEDs that are in the center on each side will be used for both the top and bottom regions. Reusing LEDs for two regions is not a problem because in this game there is never more than one of the four light regions lit at any given time.

To implement these light groups I first need to show you how to work with the LEDs in this board. Make sure your board is connected to your computer and then start the Mu editor. In the editor, open the Serial panel to enter the REPL. If you don't get a Python prompt it is because the board is executing the older **code.py** file that is stored in it. In that case press Ctrl-C to interrupt the code as shown in the previous section.

The CircuitPython release that you are running on your board comes with preinstalled libraries to control the hardware. Things such as turning lights on an off and playing sounds are easy because of these libraries. In the REPL, you can import the hardware control library for the Circuit Playground Express board as follows:

```
>>> from adafruit_circuitplayground.express import cpx
>>>
```

After you enter the above statement, the `cpx` variable can be used to control all aspects of the hardware in your board. In particular, you can control the NeoPixel LEDs with the `cpx.pixels` attribute.

Let's see what is the state of these LEDs:

```
>>> cpx.pixels
[(0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0, 0), (0, 0
>>>
```

Here you can see that the value of `cpx.pixels` is a list of 10 elements, one per LED. Each LED is represented by a 3-element tuple. The three numbers in each LED are the red, green and blue intensities, which combined can make it pretty much any color. In my case all the LEDs are set to the value `(0, 0, 0)`, which means that the LEDs are all at their minimum intensity, or in other words, turned off.

Let's turn the first LED on to a full white, which is achieved by setting the red, green and blue intensities to the maximum value of 255:

```
>>> cpx.pixels[0] = (255, 255, 255)
>>>
```

You should now have the LED referenced by the index 0 turned on!

NeoPixel LEDs are very powerful, and you may find that turning them on to full intensity like I did above is uncomfortable to your eyes. One option that you have is to use lower intensity values. For example, using `(128, 128, 128)` would reduce the intensity to 50%, and `(64, 64, 64)` to 25%.

What I think is a more convenient option, however, is to use a global brightness control provided by the board in the `cpx.pixels.brightness` attribute. This attribute can be set to a value between 0 and 1 to control the intensity of all the LEDs together. I have found that a low setting of 0.1 (10% intensitty) is the best for my eyes:

```
>>> cpx.pixels.brightness = 0.1
>>>
```

As soon as you change the brightness value you should see the effect on the LED that is on. Feel free to experiment with the global brightness control until you find the setting that you like the best.

Let's enable two more LEDs, so that we start getting familiar with how LED index numbers in the code map to the actual LEDs on the board:

```
>>> cpx.pixels[1] = (255, 255, 255)
>>> cpx.pixels[2] = (255, 255, 255)
>>>
```

Now you should have three LEDs turned on to white, the ones known in the code as indexes 0, 1 and 2. If you hold the board with the USB cable on the bottom, then these three LEDs are the ones I want to use for the bottom right color region of the the game, which according to the light diagram above is the green section. So let's make these LEDs green instead of white:

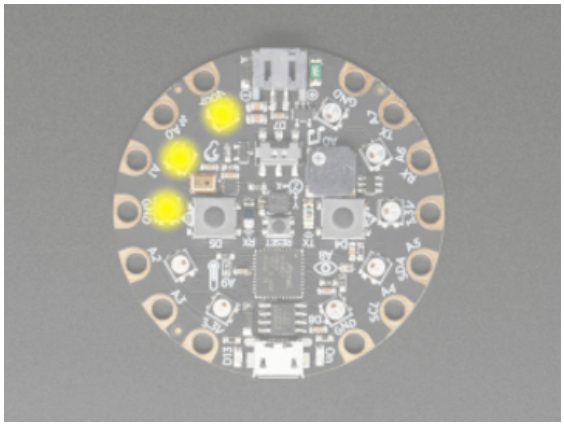
```
>>> cpx.pixels[0] = (0, 255, 0)
>>> cpx.pixels[1] = (0, 255, 0)
>>> cpx.pixels[2] = (0, 255, 0)
>>>
```

You may find it a bit tedious to have to run an almost identical sentence three times to get the green area to light up. A better way to do this is with a for-loop:

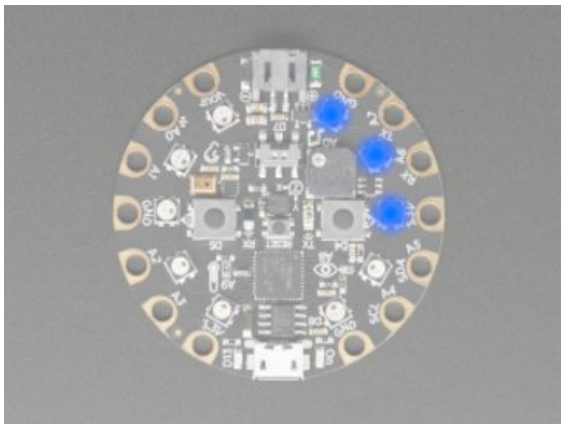
```
>>> for led in [0, 1, 2]:
...     cpx.pixels[led] = (0, 255, 0)
...
>>>
```

To enter loops like the above in the REPL you have to be aware of a trick. After you enter the `cpx.pixels` line and press enter, the cursor will go to the next line at the same indentation level, right below the `c`. To tell the REPL that the loop has ended, you have to press the Backspace or Delete keys, which will bring the cursor back to the leftmost position. At this point you can press enter to tell the REPL that you are done and want to start the execution of the loop.

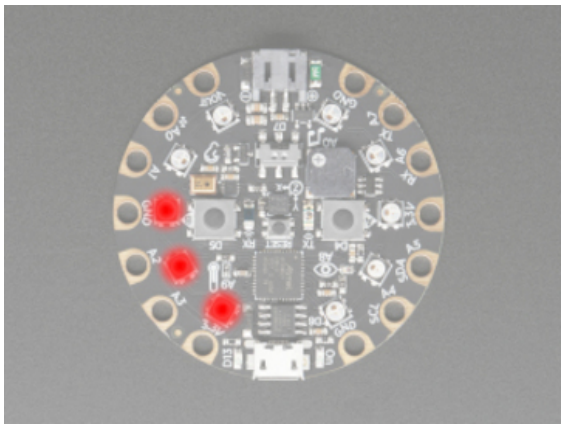
You can now continue with the task of identifying the remaining light regions and how they map to LED indexes on your own if you like, but below you can see a summary of the four light regions, with the code required to light them up with the right color:



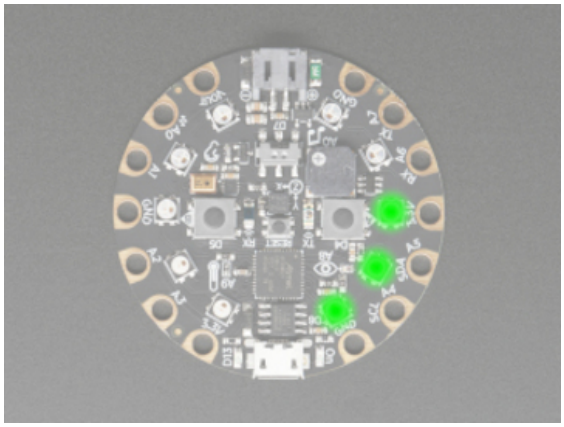
```
>>> for led in [5, 6, 7]:
...     cpx.pixels[led] = (255, 255, 0)
...
>>>
```



```
>>> for led in [2, 3, 4]:
...     cpx.pixels[led] = (0, 0, 255)
...
>>>
```



```
>>> for led in [7, 8, 9]:
...     cpx.pixels[led] = (255, 0, 0)
...
>>>
```



```
>>> for led in [0, 1, 2]:
...     cpx.pixels[led] = (0, 255, 0)
...
>>>
```

If you are trying to light up different regions, you surely noticed that when switching from one region to the next it is also necessary to turn off the old lights. This is easily accomplished by setting the intensity of the appropriate LEDs to `(0, 0, 0)`. For example, To turn off the green light region, you would do this:

```
>>> for led in [0, 1, 2]:
...     cpx.pixels[led] = (0, 0, 0)
...
>>>
```

At this point all the research necessary to implement the lights for the Simon game is complete. In the next section I will write some actual code!

Coding the Lights

What I thought would be a good way to implement the light support for this game is to have a function that I can call to light up the selected regions for a given amount of time. By calling this function repeatedly I would then achieve the effect of playing a light sequence.

Let's call this function `light_region()`. The first argument is going to be the region, which will be given as 0, 1, 2 or 3 for the four light regions in the game. The second argument is going to be the duration for the light to stay on, and for convenience I will make this argument have a default of one second. Here is how the function will look:

```
def light_region(region, duration=1):
    # TODO!
```

To implement the logic in this function, we have the problem of knowing which LEDs need to be turned on for the given `region` argument. A very convenient way to do this is to use a tuple of four tuple elements that map to the four regions. Each element of this tuple then contains the LEDs that belong to that region as a tuple.

```
REGION_LEDS = (
    (5, 6, 7),  # yellow region
    (2, 3, 4),  # blue region
    (7, 8, 9),  # red region
    (0, 1, 2),  # green region
)
```

This is great, because now I can use the expression `REGION_LEDS[region]` to know which LED indexes I need to turn on. The same concept can be applied to the region colors, using another tuple:

```
REGION_COLOR = (
    (255, 255, 0),  # yellow region
    (0, 0, 255),    # blue region
    (255, 0, 0),    # red region
    (0, 255, 0),    # green region
)
```

Having these two lists makes implementing the `light_region()` function much easier. Recall that this is the core idea of bottom-up development. We start from the trivial, and slowly move our way up to the more complex and challenging tasks using what's already built as support.

Now I'm ready to show you the first humble version of the Simon game. Before you copy the code I'll show you below, make sure that your Circuit Playground Express board is connected to your computer and your Mu editor is running with the Serial panel visible. If the main window of your editor shows some older code, click the **New** button to start a brand new file on which to write this game.

When you are ready, write (or copy and paste) the following code in your text editor window:

```
import time
from adafruit_circuitplayground.express import cpx

cpx.pixels.brightness = 0.1  # adjust brightness to your liking

REGION_LEDS = (
    (5, 6, 7),  # yellow region
    (2, 3, 4),  # blue region
    (7, 8, 9),  # red region
    (0, 1, 2),  # green region
)

REGION_COLOR = (
    (255, 255, 0),  # yellow region
    (0, 0, 255),    # blue region
    (255, 0, 0),    # red region
    (0, 255, 0),    # green region
)

def light_region(region, duration=1):
    # turn the LEDs for the selected region on
    for led in REGION_LEDS[region]:
        cpx.pixels[led] = REGION_COLOR[region]

    # wait the requested amount of time
    time.sleep(duration)

    # turn the LEDs for the selected region off
    for led in REGION_LEDS[region]:
        cpx.pixels[led] = (0, 0, 0)
```

By now you should understand most of this code. There are two imports, the `time` and `cpx` libraries. You are already familiar with `cpx`, but I haven't mentioned `time` before, which I'm going to use to implement the `duration` argument of the `light_region()` function. I have included the global brightness setting for the NeoPixels, which you can set to the level that works best for you.

The implementation of the `light_region()` function is fairly simple, because it builds on the support provided by the `REGION_LEDS` and `REGION_COLOR` lists, which I included above the function. The first part of the function turns on the three LEDs associated with the `region` passed as an argument. Note how by using the tuples this becomes almost trivial. The middle section of the function just waits for the specified duration time in seconds. This is implemented with a call to `time.sleep()`, which just halts the script for the requested time. The final section of the function sets the three LEDs for the chosen region back to `(0, 0, 0)` to turn them off.

After you enter the above code in your editor, click the **Save** button. A file dialog will open on the disk that belongs to the board. Select the name **code.py** for your file and save it.

Now click on the serial console in the bottom part of the Mu window. If you already have a Python REPL running, you need to press **Ctrl-D** to exit it and allow the board to do a soft reset to see the new code. Then enter the REPL again and test the new `light_region()` function as follows:

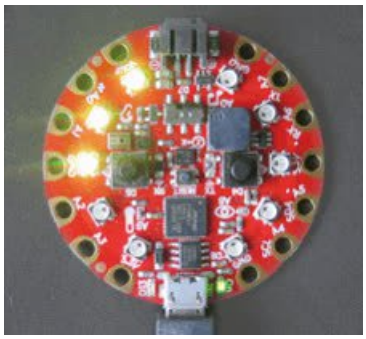
```
>>> from code import light_region
>>> light_region(0)
>>> light_region(1)
>>> light_region(2)
>>> light_region(3)
>>> light_region(0, 3)  # light up for 3 seconds instead of the default of 1 second!
```

Isn't this awesome? Now you have a high-level function that can be used to turn selected light regions just by passing the region number! This is going to simplify the task of playing a random sequence of colors, and also responding to the user input when entering the sequence.

To make the code a bit more complete, let's add a small light demo cycle. Go back to the text editor panel in Mu and enter the following at the bottom of the code, below the `light_region()` function:

```
while True:
    light_region(0)
    light_region(1)
    light_region(2)
    light_region(3)
```

Save your code, and then in the Python shell session hit **Ctrl-D** to trigger another soft reset. Because this last bit of code was added in the global scope (or in other words, outside of any functions), it executes automatically when the board restarts, and now you have a fully self-contained application that demonstrates the light functionality of the game!



Adding Sound

Each time one of the four colored buttons in the Simon game console lights up, a sound tone also plays. In this section I'm going to duplicate this functionality using the tone playback feature of the Circuit Playground Express board.

I have found a very interesting blog post by Simon Inns in which he reverse-engineered a Simon console. Among the extensive details he provides on the console internals, he includes the frequencies of the tones played for each of the four buttons. According to the article, the frequencies are as follows:

- Green: 415 Hz
- Red: 310 Hz
- Yellow: 252 Hz
- Blue: 209 Hz

This can be represented with a third tuple in the game:

```
REGION_TONE = (
    252,  # yellow region
    209,  # blue region
    310,  # red region
    415,  # green region
)
```

Playing a tone with a given frequency cannot be easier on the Circuit Playground Express. The `cpx.start_tone()` takes a tone frequency in Hertz and starts playing that tone on the on-board speaker. The tone continues to play until the `cpx.stop_tone()` is called.

With this information, I can now extend the `light_region()` function from the previous section to play tones to go along with the lights. The place where I called the `time.sleep()` function can be expanded into three parts, where first the tone is started, then the sleep call is made, and finally the tone is stopped:

```
    # play a tone for the selected region
    cpx.start_tone(REGION_TONE[region]);

    # wait the requested amount of time
    time.sleep(duration)

    # stop the tone
    cpx.stop_tone()
```

Below you can see a complete **code.py** updated with this change:

```
import time
from adafruit_circuitplayground.express import cpx

cpx.pixels.brightness = 0.1  # adjust NeoPixel brightness to your liking

REGION_LEDS = (
    (5, 6, 7),  # yellow region
    (2, 3, 4),  # blue region
    (7, 8, 9),  # red region
    (0, 1, 2),  # green region
)

REGION_COLOR = (
    (255, 255, 0),  # yellow region
    (0, 0, 255),    # blue region
    (255, 0, 0),    # red region
    (0, 255, 0),    # green region
)

REGION_TONE = (
    252,  # yellow region
    209,  # blue region
    310,  # red region
    415,  # green region
)

def light_region(region, duration=1):
    # turn the LEDs for the selected region on
    for led in REGION_LEDS[region]:
        cpx.pixels[led] = REGION_COLOR[region]

    # play a tone for the selected region
    cpx.start_tone(REGION_TONE[region]);

    # wait the requested amount of time
    time.sleep(duration)

    # stop the tone
    cpx.stop_tone()

    # turn the LEDs for the selected region off
    for led in REGION_LEDS[region]:
        cpx.pixels[led] = (0, 0, 0)


while True:
    light_region(0)
    light_region(1)
    light_region(2)
    light_region(3)
```

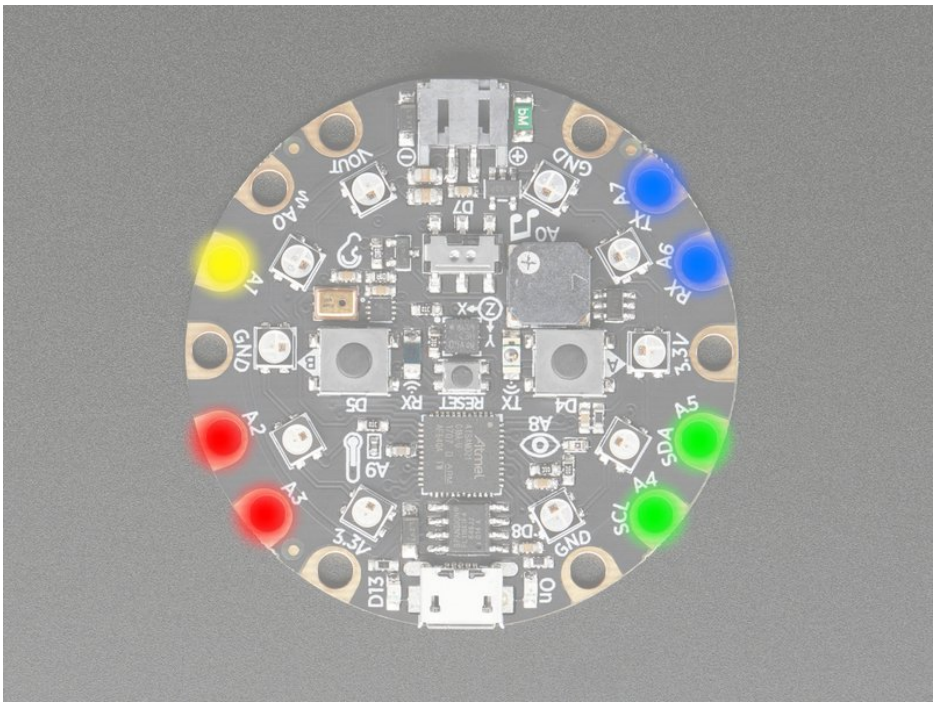
As soon as you save this version of the game and reset the board you will not only see but also hear the four lights!

Touch Input

As I make progress through my bottom-up designed game, it is now time to look at how the player will respond to the game by repeating the light sequence. In the game console, the four light regions are physical buttons that the user can press. The closest approximation to that I could come up with on the Circuit Playground Express board is to use the capacitive touch sensors.

This board comes with seven capacitive touch sensors, labeled `A1` through `A7`. You can actually see the labels printed on the board itself. You may notice that there is also an `A0` pad, but unfortunately that one is not a capacitive touch pad.

The following diagram shows how I'm going to map these seven touch sensors to the four light regions in the game:



You can see that the yellow color is at a disadvantage compared to the other regions because it only has one touch pad versus two pads each for the others. Unfortunately this is something that cannot be helped, because the design of this board has an asymmetric distribution of touch pads. If you feel it is more important to have perfectly symmetrical inputs, then you can just not use pads `A3`, `A4` and `A7`, which would leave all regions with a single touch pad positioned exactly as the yellow one.

The `cpx` module provides support for checking the state of each pad. For example, to check the state of pad `A1`, you can use the expression `cpx.touch_A1`. This is easily tested in a REPL session:

```
>>> from adafruit_circuitplayground.express import cpx
>>> cpx.touch_A1
False
```

A value of `False` indicates that the pad in question is currently not being touched. Now go ahead and touch the `A1` pad with a finger and then repeat the above statement:

```
>>> from adafruit_circuitplayground.express import cpx
>>> cpx.touch_A1
True
```

Back when I needed to map region numbers to colors or to sounds, I used lists. Now I need to map capacitive touch pads to region numbers, which is a little bit different. Since the pads have names and not numbers, this time I'm going to specify this mapping using a Python dictionary:

```
PAD_REGION = {
    'A1': 0,  # yellow region
    'A2': 2,  # red region
    'A3': 2,  # red region
    'A4': 3,  # green region
    'A5': 3,  # green region
    'A6': 1,  # blue region
    'A7': 1,  # blue region
}
```

With this data structure, once I know which pad is being touched, I can easily find the region number it corresponds to. I can now construct a function, which I'm going to call `read_region()` that returns the number of the region being touched by the player, or `None` if none of the pads are touched. Here is a first implementation of this function:

```
def read_region():
    if cpx.touch_A1:
        return PAD_REGION['A1']
    elif cpx.touch_A2:
        return PAD_REGION['A2']
    elif cpx.touch_A3:
        return PAD_REGION['A3']
    elif cpx.touch_A4:
        return PAD_REGION['A4']
    elif cpx.touch_A5:
        return PAD_REGION['A5']
    elif cpx.touch_A6:
        return PAD_REGION['A6']
    elif cpx.touch_A7:
        return PAD_REGION['A7']
```

Simple, right? Now I can replace the light demo code at the bottom of **code.py** with a slightly more interesting piece of code that waits for the user to touch one of the pads, and then lights the appropriate region:

```
while True:
    region = read_region()
    if region is not None:
        light_region(region)
```

This simple code snippet shows how powerful bottom-up development is. Since all the basic blocks of this game have now been implemented, it takes just four lines of code to make the board respond to user input in a similar way to the Simon game!

The `read_region()` function is currently implemented in a very simplistic way, it just returns the region being touched or `None`, which means that it does not wait for the player to touch a pad. An interesting improvement would be to

have this function wait for the player to select a pad for a given amount of time. This can be accomplished by wrapping the chain of if statements that do the pad checking logic in a while-loop, and having the loop exit only after the specified time has passed without the player having chosen a region:

```
def read_region(timeout=30):
    val = 0
    start_time = time.time()
    while time.time() - start_time < timeout:
        if cpx.touch_A1:
            val = PAD_REGION['A1']
        elif cpx.touch_A2:
            val = PAD_REGION['A2']
        elif cpx.touch_A3:
            val = PAD_REGION['A3']
        elif cpx.touch_A4:
            val = PAD_REGION['A4']
        elif cpx.touch_A5:
            val = PAD_REGION['A5']
        elif cpx.touch_A6:
            val = PAD_REGION['A6']
        elif cpx.touch_A7:
            val = PAD_REGION['A7']
    return val
```

Here I have expanded the function to take a `timeout` argument which specifies for how many seconds the function should wait for user input, with a default of 30 seconds. Inside the function I use the `time.time()` function to get the current time in seconds, and then on each loop iteration I check how many seconds have passed by substracting that time from an updated reading of `time.time()`. If this difference is larger than the requested timeout, then the loop exits, and the function returns `None`. But if, on the other side, the player touches one of the pads, then the function immediately returns the region touched.

The benefit of adding this loop is that now the game logic (that I have yet to write!) does not need to get complicated with keeping track of time while it waits for player input. With this solution, if `read_region()` returns `None` it means that the player took too long to answer and has lost the game.

To make sure that you have the correct code, below you can see a complete copy of **code.py** with all the work done so far:

```
import time
from adafruit_circuitplayground.express import cpx

cpx.pixels.brightness = 0.1  # adjust NeoPixel brightness to your liking

REGION_LEDS = (
    (5, 6, 7),  # yellow region
    (2, 3, 4),  # blue region
    (7, 8, 9),  # red region
    (0, 1, 2),  # green region
)

REGION_COLOR = (
    (255, 255, 0),  # yellow region
    (0, 0, 255),    # blue region
    (255, 0, 0),    # red region
    (0, 255, 0),    # green region
```

```
)

REGION_TONE = (
    252,  # yellow region
    209,  # blue region
    310,  # red region
    415,  # green region
)

PAD_REGION = {
    'A1': 0,  # yellow region
    'A2': 2,  # red region
    'A3': 2,  # red region
    'A4': 3,  # green region
    'A5': 3,  # green region
    'A6': 1,  # blue region
    'A7': 1,  # blue region
}

def light_region(region, duration=1):
    # turn the LEDs for the selected region on
    for led in REGION_LEDS[region]:
        cpx.pixels[led] = REGION_COLOR[region]

    # play a tone for the selected region
    cpx.start_tone(REGION_TONE[region]);

    # wait the requested amount of time
    time.sleep(duration)

    # stop the tone
    cpx.stop_tone()

    # turn the LEDs for the selected region off
    for led in REGION_LEDS[region]:
        cpx.pixels[led] = (0, 0, 0)

def read_region(timeout=30):
    val = 0
    start_time = time.time()
    while time.time() - start_time < timeout:
        if cpx.touch_A1:
            val = PAD_REGION['A1']
        elif cpx.touch_A2:
            val = PAD_REGION['A2']
        elif cpx.touch_A3:
            val = PAD_REGION['A3']
        elif cpx.touch_A4:
            val = PAD_REGION['A4']
        elif cpx.touch_A5:
            val = PAD_REGION['A5']
        elif cpx.touch_A6:
            val = PAD_REGION['A6']
        elif cpx.touch_A7:
            val = PAD_REGION['A7']
    return val

while True:
    region = read_region()
    if region is not None:
```

```
    if region is not None:
        light_region(region)
```

Game Logic

All of the core pieces of functionality of the game are now implemented, so it is time to start looking at the higher level logic that actually implements the game itself.

First, I know I'll be playing sequences of lights and tones. A good way to represent the growing sequence is to use a list, so I can create a `play_sequence()` function that takes a list of region numbers and reproduces the sequence by invoking the `light_region()` function for each region:

```
def play_sequence(sequence):
    duration = 1 - len(sequence) * 0.05
    if duration < 0.1:
        duration = 0.1
    for region in sequence:
        light_region(region, duration)
```

The key part of this function is in the last two lines, where a for-loop runs through the regions given in the `sequence` list one by one. But to make the game a bit more interesting I decided to make the duration of each light shorter as the sequence gets longer. I start from a duration of one second, and substract 0.05 of a second for each item that is in the sequence. The longer the sequence, the shorter the duration, so it gets harder to play. Since a very long sequence could make the `duration` variable zero or even negative, I make sure that it never goes below 0.1.

The game is going to start by playing a sequence, which initially will have a length of one. Then it will expect the player to repeat the sequence by touching the pads. The next function I'm going to add is going to be for this purpose, and I'm going to call it `read_sequence()`. This function will also take the `sequence` list, but instead of playing the color regions, it will read them from the player:

```
def read_sequence(sequence):
    for region in sequence:
        if read_region() != region:
            # the player made a mistake!
            return False
        light_region(region, 0.25)
    return True
```

You can see that here each time the player touches a region I light up that region with a short duration of 0.25 of a second. This is important, as it gives the player feedback that the input was recognized. If the user touches the wrong region, or if the read function times out and returns `None`, then the conditional will cause the function to return `False`, which will indicate that the player has lost the game. If the entire sequence is entered correctly, then the return value is `True`, and this will trigger the game logic to add one more element to the sequence and repeat the cycle.

In the event of the player making a mistake, the game needs to indicate that the game has been lost. This can be achieved with a low frequency tone. The function `play_error()` does this:

```
def play_error():
    cpx.start_tone(160)
    time.sleep(1)
    cpx.stop_tone()
```

And now, finally all the parts are in place to implement the main game logic!

To select a new color when adding to the sequence, the `random.randint()` function from CircuitPython can be used. Here is how this function works in a REPL session:

```
>>> import random
>>> random.randint(0, 3)
3
>>> random.randint(0, 3)
1
>>> random.randint(0, 3)
2
>>>
```

Each time `random.randint()` is called, a random number that is between the two arguments is generated. Each time the player reproduces the sequence correctly, a new element will be generated randomly and appended at the end of the sequence.

Here is the function `play_game()`, which implements the complete game logic:

```
import random

def play_game():
    sequence = []
    while True:
        sequence.append(random.randint(0, 3))
        play_sequence(sequence)
        if not read_sequence(sequence):
            # game over
            play_error()
            break
        time.sleep(1)
```

Isn't it amazing that the game is now so simple to write? This function starts by creating an empty `sequence` list. This is where the color regions are going to be added as the player makes progress through the game. The rest of the game is implemented inside a `while True` loop, which will run until the player makes a mistake.

The loop starts by adding a random color to the sequence. The sequence is then played by calling the `play_sequence()` function. Next the player needs to repeat the sequence, which is something that I already implemented in the `read_sequence()` function. If this function returns `False` it means that the player lost the game, so the error tone is played by calling `play_error()` and then the `break` statement causes the while-loop to exit, which in turn makes the `play_game()` function exit as well. If the player enters the entire sequence correctly, then I have a short delay of one second to give the player a short break, and then the loop will start again from the top, adding another element to the sequence and repeating the cycle.

The last snippet of code that I need is to call `play_game()` in the global scope, so that the board automatically runs the game when it is powered. Instead of just calling the function, I'm going to put it inside a `while True` loop, so that each time the player loses the game a new game is started:

```
while True:
    play_game()
```

And with this, the game is now complete!

Here is the complete code for the game. Download the code and save onto your **CIRCUITPY** drive as **code.py**.

```
import time
import random
from adafruit_circuitplayground.express import cpx

cpx.pixels.brightness = 0.1  # adjust NeoPixel brightness to your liking

REGION_LEDS = (
    (5, 6, 7),  # yellow region
    (2, 3, 4),  # blue region
    (7, 8, 9),  # red region
    (0, 1, 2),  # green region
)

REGION_COLOR = (
    (255, 255, 0),  # yellow region
    (0, 0, 255),    # blue region
    (255, 0, 0),    # red region
    (0, 255, 0),    # green region
)

REGION_TONE = (
    252,  # yellow region
    209,  # blue region
    310,  # red region
    415,  # green region
)

PAD_REGION = {
    'A1': 0,  # yellow region
    'A2': 2,  # red region
    'A3': 2,  # red region
    'A4': 3,  # green region
    'A5': 3,  # green region
    'A6': 1,  # blue region
    'A7': 1,  # blue region
}

def light_region(region, duration=1):
    # turn the LEDs for the selected region on
    for led in REGION_LEDS[region]:
        cpx.pixels[led] = REGION_COLOR[region]

    # play a tone for the selected region
    cpx.start_tone(REGION_TONE[region])

    # wait the requested amount of time
    time.sleep(duration)

    # stop the tone
    cpx.stop_tone()

    # turn the LEDs for the selected region off
    for led in REGION_LEDS[region]:
        cpx.pixels[led] = (0, 0, 0)
```

```
def read_region(timeout=30):
    val = 0
    start_time = time.time()
    while time.time() - start_time < timeout:
        if cpx.touch_A1:
            val = PAD_REGION['A1']
        elif cpx.touch_A2:
            val = PAD_REGION['A2']
        elif cpx.touch_A3:
            val = PAD_REGION['A3']
        elif cpx.touch_A4:
            val = PAD_REGION['A4']
        elif cpx.touch_A5:
            val = PAD_REGION['A5']
        elif cpx.touch_A6:
            val = PAD_REGION['A6']
        elif cpx.touch_A7:
            val = PAD_REGION['A7']
    return val

def play_sequence(sequence):
    duration = 1 - len(sequence) * 0.05
    if duration < 0.1:
        duration = 0.1
    for region in sequence:
        light_region(region, duration)

def read_sequence(sequence):
    for region in sequence:
        if read_region() != region:
            # the player made a mistake!
            return False
        light_region(region, 0.25)
    return True

def play_error():
    cpx.start_tone(160)
    time.sleep(1)
    cpx.stop_tone()

def play_game():
    sequence = []
    while True:
        sequence.append(random.randint(0, 3))
        play_sequence(sequence)
        if not read_sequence(sequence):
            # game over
            play_error()
            break
        time.sleep(1)

while True:
    play_game()
```

I hope you found this guide useful, not only to learn how to write this specific game, but also to give you some ideas on how you can use similar techniques to implement other games!