CPB - Learn CircuitPython on the CPB [Instructor Guide] - Part I
Last Updated: Nov. 30, 2019

This is Part I of "Learn CircuitPython on the CPB" material.
Find Part II at:
http://bit.ly/learn-circuitpython-on-cpb-part2

YouTube Playlist:
An accompanying YouTube playlist for Part I and Part II can be found at:
http://bit.ly/learn-circuitpython-on-cpb

Slides in Keynote:
A Google Drive containing slides in Mac Keynote can be found at:
http://bit.ly/learn-circuitpython-on-cpb-slides

Those that don't have Macs can use the web-based version at icloud.com

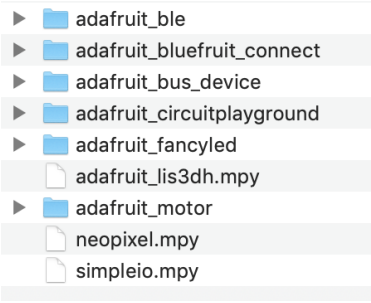
About These Tutorials
I first created this content in Fall 2019 when all 100 of my students, across two sections, received a Circuit Playground Bluefruit - ALPHA as part of their (mostly) Freshman Digital Tech: Strategy & Use core class at Boston College. We used CircuitPython 5 (alpha, then beta versions). Most of the class covers business concepts, we also have an Excel 365 topic about once a week, but I taught two 50 minute classes on CircuitPython and assigned videos above for "flipped class" sessions. Students were later tasked to create their own end-of-semester projects. For a look at what students did the prior semester (where MakeCode was introduced along with CircuitPython and the original CircuitPlayground Express was used) is at:
https://medium.com/@gallaugher/unexpectedly-awesome-hardware-programming-in-a-freshmen-level-business-school-core-class-3bdaa1bf8434

Feel free to use the content in your own classes or use this as an independent learner. Just be sure to refer back to the original source. I'd be delighted if you or your students would subscribe on YouTube (http://bit.ly/GallaugherYouTube) and follow on Twitter: @gallaugher. I also share course material online at https://gallaugher.com.

While I've taught managerial courses for over 20 years, and a full-stack iOS app development in Swift for 4 years, this is only the second semester I've taught CircuitPython as a course component, so any error catches, additional suggestions, and atta boys are most welcome.

About Python, CircuitPython, and the CPB

I assume all students have a CPB configured with the latest version of CircuitPython 5 and that their CPB has a "lib" folder containing (at a minimum) the files/folders shown below:



There is a video in the playlist describing how to configure the CPB. Adafruit's documentation describing configuration (always updated) is at: https://learn.adafruit.com/adafruit-circuit-playground-bluefruit/circuitpython

Welcome to Python, CircuitPython, and the CPB!

When you create a spreadsheet you use Excel, when you create a written document you use Word, to create Python programs we'll use an IDE (Integrated Developer Environment). There are lots of IDEs, but the one we'll use is Mu. It is very simple and creates a few special features that allow it to work well with the CPB.

You can install and download a version of Mu from:

https://codewithmu.org

Once you've got Mu running on your computer, start Mu and return to this document.

- Launch Mu if you haven't already.
- When prompted to select a mode, select "Adafruit Circuit Python"
  You may also get an error message stating
  Mu Could Not Find an Attached CircuitPython Device.
  If you get this message you likely don't have your CPB plugged into your computer. Make sure your CPB is plugged into your USB cable (it **must be a data/sync cable**, **NOT a charge-only cable**) and that this cable is plugged into your computer, then press OK.

Write some code!

The first line in Mu shows:
Write your code here :-)
The # symbol indicates that anything on the line that directly follows the space after # will be ignored when executing the program. Comments are great ways to document your Python code.

Write:

```
print("I just wrote my first Python program!")
```

Click the "**Serial**" button up top.

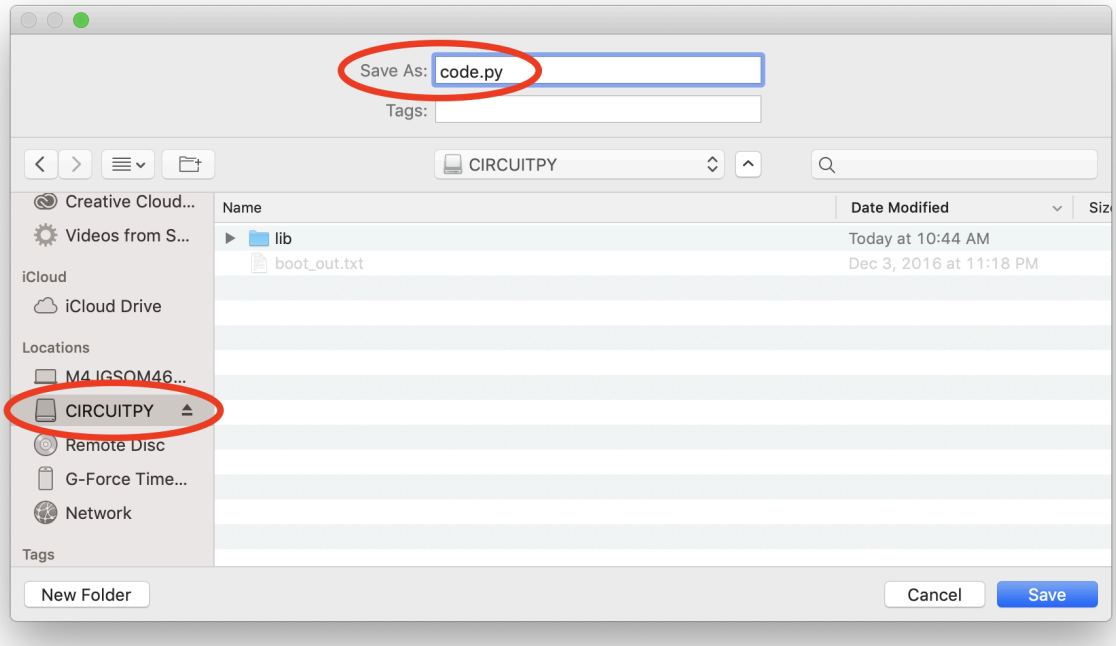
- A window will open showing:
  Auto-reload is on. Simply save files over USB to run them or enter REPL to disable.

  Press any key to enter the REPL. Use CTRL-D to reload.

- Click the "**Save**" button.

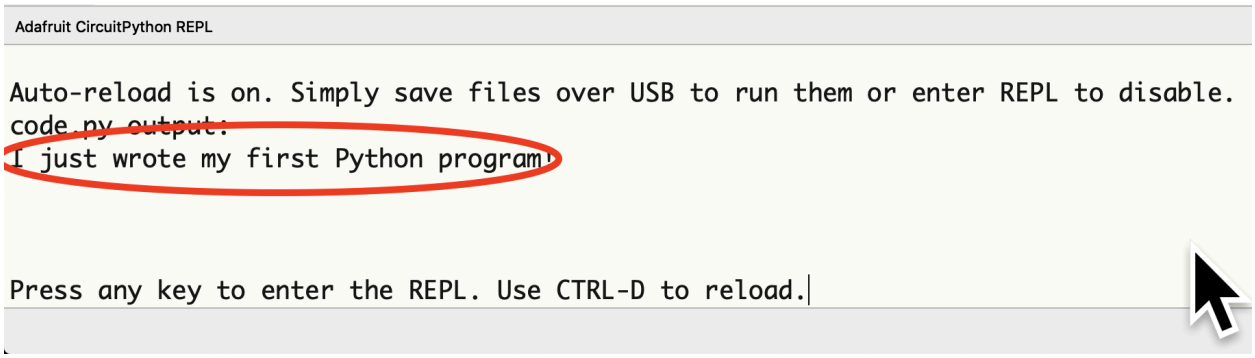
The "**Save As…**" dialog box will appear.

- Click the "**CIRCUITPY**" volume to save your work to your CPB. If, for some reason, it's not showing up as connected to your computer, just single-click the reset button in the center of the CPB and CIRCUITPY should show.



- You might not see your message printed to the serial console yet, if not just click "**Serial**" to close and open the console (this will reset what you see in the output area at the bottom of your screen), then click "**Save**" again (this will re-execute your code.py file, starting at the beginning (remember these steps, as they'll be useful when working on all future programs). After doing this, you should see the message you typed inside the print statement show up on the

console, as in the image, below:



Do a happy dance - you just wrote your first line of Python code!

Light it up!

Now let's use the build-in lights on our CPB. To use these, we'll need to import two additional libraries that will allow us to write Python code for the CPX and to use the little lights on the board (these lights are called neopixels).

You can delete the print statement that you wrote above, and replace it with these lines:

```
import board
import neopixel
```

The first line will import a library of code named **board**. We won't see this, but once we add this line, then we can write Python code specifically to interact with the CPB (as well as other boards that we might want to control with CircuitPython).

importing neopixel will allow us to use commands to control the 10 lights on the CPB, as well as lights that we could wire up to the CPB.

Before we can write code to get the CPB to light up, we need to write a line of code that will set things up. Write the line of code below:

```
pixels = neopixel.NeoPixel(board.NEOPIXEL, 10,
brightness=0.5, auto_write=True)
```

This looks a bit gnarly, but it's pretty easy to read once you get a sense of what's going on. This line lets us create a single word, **pixels**, that will refer to all 10 neopixel lights that are on our CPB.

We can then use the word **pixels** to make changes to the settings that control the lights. Since we can make changes to the settings using the word **pixels**, we say that **pixels** is a variable, since we can *vary* the settings for our lights.

> By the way, we didn't have to use the word pixels, we could have given it other names (including your own name), but since the lights are called neopixels, this is a common name used by programmers and you'll often see it used in other examples.

So if "`pixels =`" says "set up this name pixels to control the lights", we then need to state what lights we're talking about. That's the next part of this code.

```
neopixel.NeoPixel(
```

says take advantage of the neopixel library we imported (the code that lets us control lights using Python) and set up a connection to a set of neopixel lights.

Following a name with a dot, and another name is Python's way of saying: inside this thing that I know about called neopixel is another thing I known about called NeoPixel (capitalization is important - spell it wrong or with different capitalization and your code won't work).

After the parentheses in NeoPixel are a list of parameters for setting up the lights. This is how to read them.

`board.NEOPIXEL` - says you're using an external hardware board (the CPB in our case. Remember that we imported board, as well? This gave us code that recognizes hardware boards and board.NEOPIXEL says these are lights attached to this board.

`10` - is the number of lights we're referring to (there are 10 lights on our board).

`brightness=0.5` - sets brightness to 50%. Brightness takes a value from 0.0 (totally dark) to 1.0 (as bright as you can get).

`auto_write=True` - immediately applies any changes to pixels. What if it were False? Sometimes you want to make a bunch of changes to different lights, then show these changes by lighting them all up at once. Right now we'll make our changes automatically, so we'll use True.

So now we can use the word (variable) pixels to refer to all of the lights on our CPB board.

Now type the line:

```
pixels.fill((255, 0, 0))
```

This says, use the `pixels` (the lights we just set up) and fill those lights with a color that we find with the three numbers 255, 0, 0.

`.fill` is an action that we can apply to anything we create with .NeoPixel(These actions are called methods in Python, and dot notation allows us to apply the action (fill) to the object (pixels).

This action (known as a method in Python) takes a single input parameter. method inputs are included inside parentheses.

The input is this:

(255, 0, 0)

The additional parentheses says treat these three numbers as one thing.
The three numbers represent a color.
Computers can represent any color as an input of three values Red, Green, and Blue (RGB). The first number refers to Red, the second Green, and the third Blue. Numbers go from zero (none of that color) to 255 (all of that color). What color do you think will show with this command?

```
pixels.fill((255, 0, 0))
```

Let's try out our code and see!
Press the "Save" button in Mu. Save will save and execute your code, right away. Look at your CPX. Cool! All red!

Challenge:
Now modify your code to get the lights to show up as all green.
Then try to set them to all blue.

You can vary the inputs to mix Red, Green, and Blue to create different colors. If you search online you'll find lots of RGB color-picker tools. There's a simple one at:

https://rgbcolorcode.com/

Neopixels are a bit limited, and don't offer perfect representations of the colors that you'll see, but you can get close.

Try finding settings that work to display Yellow, Orange, Purple, White, and Black (when you get better at CircuitPython and want to kick up your LED

game you can investigate the FancyLED library. More info at: https://learn.adafruit.com/fancyled-library-for-circuitpython/overview).

Using time.sleep to Flash the Lights:

Our code stops running once we hit the last line. If lights are on at that point, then the lights will remain on in their set color until power is removed.

If you want to turn the lights off, you can type a command to fill the lights with the color (0, 0, 0). What happens if our code looks like this?

```
import board
import neopixel

pixels = neopixel.NeoPixel(board.NEOPIXEL, 10,
brightness=0.5, auto_write=True)

pixels.fill((255, 0, 0))
pixels.fill((0, 0, 0))
```

This really quickly flashes red, then turns off all of the pixels (the equivalent of turning them black).

We can control the time in between turning on red lights, then turning them off, by using a .sleep command. .sleep isn't built into CircuitPython, but we can get this command by importing a library named time, so add the following line below our first two import statements:

```
import time
```

> **Pro Tip:** It's critical to include the above statement, otherwise your CPB won't know about the time library & the serial monitor will show

an error like this:

```
NameError: name 'time' is not defined
```

Then add the following line in between our `pixels.fill` statements:

```
time.sleep(0.5)
```

This line of code says: use the time library, which has a sleep method. The sleep method takes a single input, a number that refers to the length, in seconds, that your program should "sleep" before moving on to the next line. 0.5 says sleep for half a second, which should pause the super-fast flash of red so that it takes half a second.

Look at your CPB while clicking Save. You should see the lights light up for half a second because you turned them on, waited half a second, then turned them off.

Use while True: for continuous blinking

When working with boards like the CPB, we often don't want our code to stop - we'd like it to keep running. To do this, we'll add a loop to our code. To create a continuous or infinite loop that repeats itself over and over, use this command. Remember, Python is very picky about letter cases, punctuation, and spacing, so be sure it is entered exactly as you see it below.

```
while True:
```

If you press return after the : in the line above, you'll see Mu will automatically indent your code four spaces. Anything indented after the while True: statement will execute once, then when we get to the end of the indented code, we'll loop back to the first line after `while True:` and repeat.

The repeat happens forever. while loops essentially can be read as "while something is true, do whatever is indented after the :" Python (and most other programming languages) have values that can be set to either True or False, but here we are essentially saying while True is True (which is always the case), repeat doing the stuff that is indented. Programmers call this an "infinite loop".

> **Fun fact**: Apple Computer's original Cupertino California campus had the address "1 Infinite Loop". I've visited several times, and many of my former students work there.

So now our code should look like this:

```
import board
import neopixel
import time

pixels = neopixel.NeoPixel(board.NEOPIXEL, 10,
brightness=0.5, auto_write=True)

while True:
    pixels.fill((99, 0, 0))
    time.sleep(0.5)
    pixels.fill((0, 0, 0))
```

Save and run and see what happens.

Hmmm… it shows mostly red with a very brief flash. What do you think is going on?

The `while True` loop essentially says:

fill the lights with red
wait half a second
turn them off (by setting R, G, B to zeros)

and since we're at the end of our block of indented code, we loop back up to the : and repeat.
Did we wait after turning the lights off before turning the lights back on again at the start of the `while True` loop? Nope. So how much time is between turning lights off and turning them back to red? Pretty much nothing. The super-fast on-off flash you see is essentially a quick power cut before going back to fill red.

> **Pro Tip**: Any code that follows the `while True` loop and is not indented will never execute. This is because the `while True` loop will repeat once it hits the last statement below it that is indented at least one level to the right its own indent level. For example, in the code block below, the print statement will never be executed. Once the statement `pixels.fill((0, 0, 0))` is reached, the while True loop repeats again, re-starting at its first indented statement, which is `pixels.fill((99, 0, 0))`. Since the `print` statement is indented at the **same level** as `while True` and is **after** `while True`, the print statement will never execute. It's outside of the infinite loop.
>
> ```
> while True:
>     pixels.fill((99, 0, 0))
>     time.sleep(0.5)
>     pixels.fill((0, 0, 0))
> print("I will never be printed!")
> ```

Challenge: Stay Dark for Half a Second

Try to get the lights lights to stay dark for half a second after they turn off, so that the lights flash half a second on, then half a second off.

Solution: Stay Dark for Half a Second

```
while True:
    pixels.fill((255, 0, 0))
    time.sleep(0.5)
    pixels.fill((0, 0, 0))
    time.sleep(0.5)
```

Challenge: School Colors

Now see if you can get the lights to flash in what are roughly your school colors. Many universities have a chant "We Are" (pause two seconds) "<school name or mascot>". For example, at Boston College we say:

"We Are"
pause two beats
"BC"
pause two beats, then restart

Modify your code so that you do two rapid flashes in the first of your two colors for "We Are"
Then pause for two seconds
Then flash then do two rapid flashes in the second of your school colors
Use the .sleep command to insert the proper number of pauses so the flashes will continue to loop with timing roughly in synch with your school's chant.

Solution: School Colors

Note: Your sleep times may be a bit different, but if you feel good about the timing of your flashes, then you've got school spirit.

```
import board
import neopixel
import time

pixels = neopixel.NeoPixel(board.NEOPIXEL, 10,
brightness=0.5, auto_write=True)

while True:
    # light up maroon
    pixels.fill((128, 0, 0))
    # wait 0.15 seconds while these lights are on
    time.sleep(0.15)
    # then turn the lights off
    pixels.fill((0, 0, 0))
    # and keep them off for .1 second
    time.sleep(0.1)
    # then light up maroon a second time
    pixels.fill((128, 0, 0))
    # wait 0.15 seconds while these lights are on
    time.sleep(0.15)
    # then turn the lights off
    pixels.fill((0, 0, 0))
    # and keep them off for 1.25 seconds
    time.sleep(1.25)

    # turn all lights gold
    pixels.fill((255, 213, 0))
    # and keep them on for 0.15 seconds
    time.sleep(0.15)
    # turn the lights off
    pixels.fill((0, 0, 0))
    # and keep them off for 0.1 second
```

```
    time.sleep(0.1)
    # turn the lights gold again
    pixels.fill((255, 213, 0))
    # and keep them on for 0.15 seconds
    time.sleep(0.15)
    # turn the lights off
    pixels.fill((0, 0, 0))
    # and keep them off for 1.25 seconds
    time.sleep(1.25)
    # after waiting 1.25 seconds, loop back to the first
line after the while True: statement
```

Lists - Turning on Specific Lights

Python has a data structure known as **lists**. You can consider a variable that is a list to be like a big box that inside of it has little boxes. The boxes are indexed (numbered), beginning with zero.

> **Pro Tip:** (lots of things in computing start counting at zero instead of one. This is often referred to as **zero indexing**).

Our variable named `pixels` is actually a list. You can refer to individual elements in the list named pixels by putting an index value in square brackets.

`pixels` <- refers to all 10 of the lights
`pixels[0]` <- refers to the first light
`pixels[9]` <- refers to the last light

So how do we light up an individual light? While the `.fill` method will fill in all of the lights a specific color, you can set a light to a single color by using the equal sign to assign the color. For example:

```
pixels[4] = (0, 255, 255)
```

will light up the fifth light (remember, the first one is light zero) with greenish blue light.

Challenge: Half and Half
Write code to light up the first five lights in red, and the last five in blue.

Solution: Half and Half

Note - we didn't need a while True loop here since we just want to turn the lights on, and then we're done. The lights will remain on as long as the CPB is plugged in to your computer or powered with a battery pack. And since code.py has been saved to the CPB, any time the CPB receives power, the last version of code.py on the CPB will begin to execute.

```
# Half & Half
import board
import neopixel
import time

pixels = neopixel.NeoPixel(board.NEOPIXEL, 10, brightness =
0.5, auto_write = True)

pixels.fill((0, 0, 0))

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

```
pixels[3] = (255, 0, 0)
pixels[4] = (255, 0, 0)
pixels[5] = (0, 0, 255)
pixels[6] = (0, 0, 255)
pixels[7] = (0, 0, 255)
pixels[8] = (0, 0, 255)
pixels[9] = (0, 0, 255)
```

Challenge:
Now return to using the `while True` loop. Write a program that will turn the first button blue, then wait one tenth of a second, turn the next light blue, wait a tenth of a second, repeating this pattern until all 10 lights are blue. Then turn all lights off and wait a tenth of a second and repeat the pattern from the beginning.

For Loops

So far we've used `while True`: to loop continuously. What if we want to go through a loop only a certain number of times? We can use a "`for`" loop.

The basic for loop looks like this:

```
for VALUE_NAME in range(NUMBER):
    INDENTED_CODE_HERE
```

The `VALUE_NAME` can be any valid variable name in Python, but you'll most often see names like `i`, `x`, or `index`.

`range(NUMBER)` will repeat the code indented under the loop NUMBER times.

Any statements indented after the colon ending the `for in range:` statement will be repeated NUMBER times, with `VALUE_NAME` starting

at 0 during the first pass through the loop, then becoming 1 during the second pass, 2 during the third pass, … and ending at NUMBER-1.

What do you think this code would do?

```
for i in range(10):
    print(i)
```

If you run it with the serial monitor open, you'll see:

0
1
2
3
4
5
6
7
8
9

Now let's use a for loop to light up our neopixel lights, one by one, in the color violet, waiting half a second after each light is lit. Use this code:

```
for index in range(10):
    pixels[index] = (255, 0, 255)
    time.sleep(0.5)
```

Now let's read through this code line-by-line. The statement:

```
for i in range(10):
```

says

- create a value named `i`. For the first pass through the loop, `i` will equal zero.
- Perform all of the statements that are indented after the for loop.

    - The first time through, set pixels[0], since i = 0, to violet
    - wait half a second before continuing
- then repeat the two lines under the for loop, but for this second pass through, `i` equals 1.
    - set the second light, pixels[1], to violet
    - wait half a second before continuing
- repeat the indented code a third time, but this time set `i` equal to 2.
    - set the third light, pixels[2], to violet
    - wait half a second before continuing
- and keep repeating until you've gone up to, but not including 10 (meaning that we go through the statements ten times, starting at zero and ending at nine, but never reaching 10).

Try it out. This will light up each light, one after the other, with a two-tenths delay between them.

Challenge: One Blue at a Time (with a loop)
The lights remain blue. How would you turn off all lights and wait one tenth of a second before beginning to turn the lights on, as blue, one by one?

Solution: One Blue at a Time (with a loop)
Pay attention to the indentations below. Since the two lines after the `for` loop are indented, those repeat 10 times (from zero to nine).
Then, since the last two lines above are indented at the **same level** as the `for` loop, they are not part of the `for` loop. They are, however, part of the `while True`: loop because they are also indented after the `while True`: statement.

So inside `while True`:, the two lines after the for statement execute 10 times in a row, with `i` going from 0 to 9.
Then the pixels fill black (0, 0, 0).
Then we wait half a second

Then we repeat the `while True:` loop over again, restarting the for loop with i = 0.

```
while True:
    for i in range(10):
        pixels[i] = (0, 0, 255)
        time.sleep(0.1)
    pixels.fill((0, 0, 0))
    time.sleep(0.1)
```

Challenge: On one at a time, Off one at a time
Write some more code so that after you turn all lights to blue (above), turn them all off, starting at light 0 and going through light 9, then repeat (turning them all blue, then all off, one by one).

Solution: On one at a time, Off one at a time

```
while True:
    for i in range(0, 10):
        pixels[i] = (0, 0, 255)
        time.sleep(0.2)
    for i in range(0, 10):
        pixels[i] = (0, 0, 0)
        time.sleep(0.2)
```

<u>Understanding For Loops with Two and Three Parameters:</u>

Ranges in python can take up to three parameters:

```
range(start, stop_before_reaching, step)
```

If you know you want a `range` from 0 through n-1, you can simply use the range statement:

```
range(n)
```

For example, replace the `while True:` loop, in your code with the `for` loop below, then click the "Serial" button to show the Serial monitor, and click "Save" to save and run your program.

```
for i in range(11):
    print(i)
```

You'll see this output in the serial monitor:

```
Auto-reload is on. Simply save files over USB to run them or enter REPL to disable.
code.py output:
0
1
2
3
4
5
6
7
8
9
10


Press any key to enter the REPL. Use CTRL-D to reload.
```

The numbers print from 0 through 10 because with only one input the range starts at zero and goes up to but never reaches 11.

But what if you want to start counting at 1 instead of zero? You'd use this code:

```
for i in range(1, 11):
    print(i)
```

The first parameter is the start value, the second value is where the range will stop before reaching this value.

What if you want to count backwards? That's what the third range parameter, the step, is for.

```
for i in range(10, -1, -1):
    print(i)
```

This will do a "Blast-off" style countdown, starting at 10 and stopping at zero (it never reaches -1), and since the step is -1, it counts backward. You could also set the step to other numbers. 5 would count by 5, -10 would count backwards by tens.

Brightness

Each NeoPixel object has a `brightness` property that you can set from no brightness (0.0) to full brightness (1.0). Try this code:

```
while True:
    pixels.fill((0, 0, 255))
    pixels.brightness = 0.25
    time.sleep(0.1)
    pixels.brightness = 0.5
    time.sleep(0.1)
    pixels.brightness = 0.75
    time.sleep(0.1)
    pixels.brightness = 1.0
    time.sleep(0.1)
```

You should see your CPXb get brighter from 25% to 100% intensity, then repeat, with a one tenth of a second wait between brightness changes.

Challenge: Brightness Pulse

Use a `for` loop to pulse from 0 to 100% intensity, going up 1% at a time, with 0.01 seconds wait in between each increase. Once full intensity is reached, repeat again starting at 0%.

Hints:

`for` loop ranges only take whole integers, so you can't use numbers with decimal points, but the `brightness` property expects a decimal from 0.0 (totally dark) to 1.0 (100% brightness).

You can perform math on any number passed into `brightness`. The parameters for performing basic math functions are the same you'd find in Excel (as well as for most other programming languages), with the symbols `+, -, *, / used` for addition, subtraction, multiplication, and division.

Solution: Brightness Pulse

In the code below the first for loop executes 101 times, from i = 0 to i = 100.
The first time through, i = 0 and pixels.brightness is set to 0/100 = 0.0
The second time through, i = 1 and pixels.brightness is set to 1/100 = 0.01
The third time through, i = 2 and pixels.brightness is set to 2/100 = 0.02
…
The 100th time through i = 99 and pixels.brightness is set to 99/100 = 0.99
The 101st time through i = 100 and pixels.brightness is set to 100/100 = 1.0

The second for loop also executes 101 times, but starting at 100 and ending at 0 (remember, it goes down to, but never reaches -1), and it counts down, decreasing by 1 each time.

```
while True:
    pixels.fill((0, 0, 255))
    for i in range(0, 101): # go from 0 to 100
        pixels.brightness = i/100
        time.sleep(0.01)

    for i in range(100, -1, -1):
        pixels.brightness = i/100
        time.sleep(0.01)
```

Constants for Colors

It may make your code easier to read if you can assign a name to the three digit value for a given color. We could do this with a variable, but since once we call something red, we don't want to change what red means, so we'll

indicate our intention not to change the value by writing it in all capital letters. While variables can be changed (programmers sometimes call the ability to change, mutable), unchanging or immutable values are called constants.

Put the following code before your while True: statement.

```
RED = (255, 0, 0)
ORANGE = (255, 50, 0)
YELLOW = (255, 165, 0)
GREEN = (0, 255, 0)
CYAN = (0, 255, 255)
BLUE = (0, 0, 255)
INDIGO = (50, 0, 255)
VIOLET = (75, 0, 130)
```

You can now refer to the color name instead of having to enter the value. For example:

```
pixels.fill(GREEN)
```

is the same as

```
pixels.fill((0, 255, 0))
```

Putting colors into a list

If we want to create a list of all of our colors created above, we can write it as:

```
colors = [RED, ORANGE, YELLOW, GREEN, BLUE, INDIGO, VIOLET]
```

There are 7 values in the list named colors. RED is value `colors`[0] and violet is value `colors`[6].

Challenge: A Rainbow of Fills

Write a program that will go through all of the colors in the list colors, changing all lights to that color, then waiting 0.5 seconds to move on to the next color. When all of the colors have been used, repeat the process.

Solution: A Rainbow of Fills

```
import board
import neopixel
import time

pixels = neopixel.NeoPixel(board.NEOPIXEL, 10, brightness =
0.5, auto_write = True)

RED = (255, 0, 0)
ORANGE = (255, 50, 0)
YELLOW = (255, 165, 0)
GREEN = (0, 255, 0)
CYAN = (0, 255, 255)
BLUE = (0, 0, 255)
INDIGO = (50, 0, 255)
VIOLET = (75, 0, 130)

colors = [RED, ORANGE, YELLOW, GREEN, BLUE, INDIGO, VIOLET]

while True:
    for i in range(7):
        pixels.fill(colors[i])
        time.sleep(0.5)
```

How big is my list? Ask len

The code above works only for a list that has 6 elements in it. If we have fewer than 6 elements in colors and try to run our code, the program will crash with an error saying "`List index out of range`". If we have more than 6 elements in colors, we'll only show the first seven colors (since our range is zero through 6).
Our code would be more flexible if we could set a range that started at zero and continued through the last element in the range. We can do that by using the len function.
`len(LIST_NAME)` will return an integer with the count of elements in a list.

```
len(colors)
```

will return 7 in our code, since our colors list has 7 elements in it.

Challenge: Looping through All Colors, Regardless of List Size, Ending in BLACK
Modify the code that you wrote in the prior challenge so that you loop through a range for any size color list that contained one or more elements.

Once you've done this, add another color, BLACK, as zeros for all three RGB values, and add BLACK to the end of the list. Run your code. If it works, you should see all lights go out for half a second after the VIOLET color shows.

Solution: Looping through All Colors, Regardless of List Size, Ending in BLACK

```
import board
import neopixel
import time

pixels = neopixel.NeoPixel(board.NEOPIXEL, 10, brightness =
0.5, auto_write = True)
```

```
RED = (255, 0, 0)
ORANGE = (255, 50, 0)
YELLOW = (255, 165, 0)
GREEN = (0, 255, 0)
CYAN = (0, 255, 255)
BLUE = (0, 0, 255)
INDIGO = (50, 0, 255)
VIOLET = (75, 0, 130)
BLACK = (0, 0, 0)

colors = [RED, ORANGE, YELLOW, GREEN, BLUE, INDIGO, VIOLET,
BLACK]

while True:
    for i in range(len(colors)):
        pixels.fill(colors[i])
        time.sleep(0.5)
```

Using print statements to understand your code:

Our code should be working perfectly, but sometimes we want to take a look at the values in our code at a particular point in time. Adding print statements can help, and programmers frequently use print statements to diagnose errors or to better understand new concepts.
The while True: loop below will print the i value and the colors[i] value to the Serial output each time a color is changed. If you want to stop an executing program, remember you can type Control-C.

```
while True:
    for i in range(len(colors)):
        pixels.fill(colors[i])
        print("i = ", i)
        print("colors[i] = ", colors[i])
```

```
        time.sleep(0.5)
```

Output from this running code looks like this:

```
i = 0
colors[i] = (255, 0, 0)
i = 1
colors[i] = (255, 50, 0)
i = 2
colors[i] = (255, 165, 0)
...
```

The text inside of double quotes in the print statements above is printed as a label and could include any text. Values without the text are variable names, constants, or expressions and will print the contents of these values. Commas must be used to separate multiple items inside the parentheses of a print statement.

Buttons and Conditions with the if statement:

Now let's respond to button presses. To do this we'll need to import another library. Add this line after the last import line at the top of your Python code:

```
import digitalio
```

In the same way we created a pixels variable above to refer to all of our lights, we'll create a variable to refer to the "A" button on the CPB

```
button_A = digitalio.DigitalInOut(board.BUTTON_A)
```

This line uses the digitalio library we just imported and creates a DigitalInOut reference to the board's BUTTON_A. DigitalInOut refers to messages that can go into the board, or that are sent outward from the board. When we press the button we can consider that as being an input, so the next line says that button_A will be listening for inputs called Pull.DOWN, which means pressing the button down.

```
button_A.switch_to_input(pull=digitalio.Pull.DOWN)
```

We check to see if the button is being pressed by using an if statement. First enter the code below inside the while True: loop:

```
    if button_A.value:  # button_A is pushed
        pixels.fill((255, 0, 0))
```

The variable we created above, named button_A has a property named `.value`, and we can read this value in our code as:

```
button_A.value
```

We say that value is a property of the `DigitalInOut` that is named button_A

This value is either True or False. Values that can only be True or False are referred to as boolean values (named after the mathematician George Boole).

An if statement in Python checks to see if something is True. If it is, it performs all of the code that is indented after the if statement. If it is not True (meaning it is False), then it skips the indented code and continues on with the first line after the indented code.

Challenge: Button_A on Button_B off

The code above isn't going to do much. When the A button is pressed on your CPB, the lights will all turn RED, but then they'll stay that way since there is no code that changes the lights for any other condition. Using what you've learned above, create a new DigitalInOut object named button_B (hint, board can find this by referring to `board.BUTTON_A`). Then add another if statement that will turn all of the lights BLUE if button_B is pressed.

The completed program should show Red with A is pressed and Blue when B is pressed.

Solution: Button_A on Button_B off

Note: the code below doesn't show the "colors" and neopixels code from above. You don't have to delete this code, but we're not using it in this example.

```
import board
import neopixel
import time
import digitalio

pixels = neopixel.NeoPixel(board.NEOPIXEL, 10,
brightness=0.5, auto_write=True)
RED = (255, 0, 0)
ORANGE = (255, 50, 0)
YELLOW = (255, 165, 0)
GREEN = (0, 255, 0)
CYAN = (0, 255, 255)
BLUE = (0, 0, 255)
INDIGO = (50, 0, 255)
VIOLET = (75, 0, 130)
BLACK = (0, 0, 0)

colors = [RED, ORANGE, YELLOW, GREEN, BLUE, INDIGO, VIOLET,
BLACK]

# set up button_A
button_A = digitalio.DigitalInOut(board.BUTTON_A)
button_A.switch_to_input(pull=digitalio.Pull.DOWN)
```

```
# set up button_B
button_B = digitalio.DigitalInOut(board.BUTTON_B)
button_B.switch_to_input(pull=digitalio.Pull.DOWN)

while True:
    if button_A.value:  # button_A is pushed
        pixels.fill(RED)
    if button_B.value:  # button_B is pushed
        pixels.fill(BLUE)
```

Extending if with elif and else:

What if we want to turn on a light when the button is being held down, but turn it off when the button is released? The if statement can be expanded using an "`else`" condition to list a statement or set of statements to be performed when the if statement is False. For example:

```
while True:
    if button_A.value:  # button_A is being pushed
        pixels.fill(RED)
    else: # if button_A is NOT pushed
        pixels.fill(BLACK)
```

will turn all the lights to red if button_A is pressed (and when button_A is pressed, button_A.value will equal True).
But if button_A is not being pressed - i.e. button_A.value equals False, then all of the lights are turned off - i.e. set to color (0, 0, 0).
Note that the else statement is indented at the same level as the if statement.
Also note that both the if and else statements end in a colon (:). If you don't enter the code exactly like this, then it won't work.

Challenge: A RED, B BLUE, else BLACK

Using the example code above, also add code so that button_B only shows blue when the button is being pressed down. If written properly, this program should never show any lights unless a button is being held down. Hold down button A and you should see all lights as red. Hold down button B and you should see all lights as blue.

Solution:

You might have written code that looks like this. It works, but if you look closely, you might see that the red or blue lights flicker very quickly when held down.

```
import board
import neopixel
import time
import digitalio

pixels = neopixel.NeoPixel(board.NEOPIXEL, 10,
brightness=0.5, auto_write=True)
RED = (255, 0, 0)
ORANGE = (255, 50, 0)
YELLOW = (255, 165, 0)
GREEN = (0, 255, 0)
CYAN = (0, 255, 255)
BLUE = (0, 0, 255)
INDIGO = (50, 0, 255)
VIOLET = (75, 0, 130)
BLACK = (0, 0, 0)

colors = [RED, ORANGE, YELLOW, GREEN, BLUE, INDIGO, VIOLET,
BLACK]

# set up button_A
```

```
button_A = digitalio.DigitalInOut(board.BUTTON_A)
button_A.switch_to_input(pull=digitalio.Pull.DOWN)

# set up button_B
button_B = digitalio.DigitalInOut(board.BUTTON_B)
button_B.switch_to_input(pull=digitalio.Pull.DOWN)

while True:
    if button_A.value:  # button_A is pushed
        pixels.fill(RED)
    else:
        pixels.fill(BLACK)

    if button_B.value:  # button_A is pushed
        pixels.fill(BLUE)
    else:
        pixels.fill(BLACK)
```

Why the flicker? The code evaluates the first if statement. If A is held down, then the lights all turn red. The else condition is then skipped. But then the statement `if button_B.value:` is encountered. Since button B is not held down, the execution skips to the else statement, where the lights are briefly turned off (since button_B.value) is not True. That creates the flicker. It'll be more pronounced if we add a time.sleep statement after the last if else statement, and indented at the same level as the if else statements, like this.

```
    while True:
        if button_A.value:  # button_A is pushed
            pixels.fill(RED)
        else:
            pixels.fill(BLACK)
```

```
        if button_B.value:  # button_A is pushed
            pixels.fill(BLUE)
        else:
            pixels.fill(BLACK)

        time.sleep(0.5)
```

If you hold down button A, this code:

- flashes red
- skips down to the if button_B.value statement
- skips to this statement's else condition
- turns off the red (with almost no pause between red flash and off)
- then waits a half second after the pause.

This is quick flash red, turn off for .5 seconds, repeat

Why do things look different if you press button B?
After the while True loop:

- the first if statement skips to its else condition, which turns off the lights
- the next if statement checks to see if B is being pressed. It is, so it turns the lights blue, skipping the else condition and re-starting the while True: loop

This is: off, flash blue, repeat
time.sleep(0.5) happens when the lights are already blue, so it looks almost like they're permanently blue. The very brief flash occurs when the while True: loop is restarted. Since button A isn't being held down, we skip to the else that immediately turns off all the lights, but then immediately goes to the next if statement and turns them back on again.

Using elif to perform only one task from a list of many options:

So what we want to eliminate the flash by doing the following:
Show red lights when button A is pressed
Show blue lights if button B is pressed
or if neither is pressed, then turn the lights off.

Instead of having two separate if else statements, we can have one statement that looks at each of the conditions mentioned above. This is what it looks like:

```
while True:
    if button_A.value:  # button_A is pushed
        pixels.fill(RED)
    elif button_B.value: # button_B is pushed
        pixels.fill(BLUE)
    else: # neither button_A nor button_B is pushed
        pixels.fill(BLACK)
```

This statement has three conditions, broken up as statements starting with

- if
- elif
- and else

The code first looks to see if button A is being pressed, by evaluating

```
if button_A.value:
```

If this is not true, then the code after this statement is skipped, and execution jumps to the statement:

```
elif button_B.value:
```

You can read `elif` as ***else if*** it is true that button_B.value is equal to True, which is our way of checking to see if button B is being held down.
If it is, we turn all of the colors blue and skip the rest of the code in the if elif else statement, but if it's not true that button A is being held down and it's

also not true that button B is being held down, then the execution of our program jumps to:

```
    else:
        pixels.fill(BLACK)
```

The else here means if none of the conditions we checked for above are true, then do the line or lines that are indented after else: If we get here, no buttons were pressed, so we turn off all of the lights.

Block Comments:

Let's temporarily comment out the code for the if, elif, else loop above.

While you can put a pound sign and a space before each line of code to comment it out, like so:

```
while True:
    # if button_A.value:  # button_A is pushed
    #     pixels.fill(RED)
    # elif button_B.value:
    #    pixels.fill(BLUE)
```

You can use type a double quote character three times in a row (with no spaces) to effectively say "ignore everything after this until you see another double quote character typed three times in a row. So the text below does the same thing, but you only have to type:

```
    """
```

before and after the block of code that you want to comment out. This is handy when trying to temporarily ignore some code you wrote without having to delete and retype it.

```
while True:
    """
    if button_A.value:  # button_A is pushed
```

```
        pixels.fill(RED)
    elif button_B.value:
        pixels.fill(BLACK)
    """
```

Use """ to block out all of the code you've written after while True: Your code should look like this:

```
while True:
    """
    if button_A.value:  # button_A is pushed
        pixels.fill(RED)
    elif button_B.value:
        pixels.fill(BLUE)
    else:
        pixels.fill(BLACK)
    """
```

<u>Using Compound Conditionals to detect if both buttons are being held down:</u>

Adding an "and" between values that can be True or False will execute the statements after it, only if ***both*** statements are True

Example: This code will check to see if both button_A and button_B are both being held down at the same time. If so, the lights named pixels will light up GREEN.

```
while True:
    if button_A.value and button_B.value:
        pixels.fill(GREEN)
    elif button_A.value:
        pixels.fill(RED)
```

```
elif button_B.value:
    pixels.fill(BLUE)
else:
    pixels.fill(BLACK)
```

By the way

- it's possible to check if one part of the statement is True ***or*** another part of the statement is true by using the keyword `or` instead of `and`.

Challenge:
What do you think will happen if we change the word `and` in the code above to the word `or`? Take a guess, then try it out!

```
if button_A.value or button_B.value:
    pixels.fill(GREEN)
else:
    pixels.fill((0, 0, 0))
```

Here is are "Truth Tables" to help explain how the boolean logic of `and` and `or` works.

expression_A:	**expression_B:**	**expression_A** and **expression_B**
True	True	True
True	False	False
False	True	False
False	False	False

expression_A:	**expression_B:**	**expression_A** or **expression_B**
True	True	True
True	False	True
False	True	True
False	False	False

Another way to think of this is that the statement

    if A and B:

is only True if both A and B are both true, while the statement:

    if A or B:

is only False if neither A nor B is true.

This is the end of Part I in this series, Part II is at:
http://bit.ly/cpb-circuitpython-exercises