Sensors for Journalism

Matt Waite

2020-06-18

Contents

Introduction

Most of the data journalism done throughout the world uses data collected by the government. The problem with that is twofold: That means data journalists are reliant upon the government to do their journalism, and what happens when the government doesn't collect the data.

There are ways for journalists to create their own data – surveys, keying in data from documents or, for well funded organizations, hiring workers to collect it. Another way, which is lightly explored but potentially powerful, is using off-the-shelf sensors to collect that data.

Some major milestone projects for sensors in journalism:

- Cicada Tracker: A WNYC project that made ground temperature sensors and a means for people to report that temperature to track the emergence of 17-year cicadas. Hundreds of volunteers built sensors and reported data to the project in 2013.
- What's in the Air?: The School of Journalism and Media Studies at San Diego State University used air quality sensors to report on air pollution in their city in 2014-2015.
- Code Red: The University of Maryland's Philip Merrill College of Journalism partnered with NPR to report on inequality in climate change, specifically measuring differences in temperatures inside people's houses. They built temperature and humidity sensors to show what it's like to live in Baltimore's rowhouses during a hot summer.

What all three projects combine is some simple electronics, inexpensive sensors, basic programming and a giant bucket of creativity. The electronics and the programming are within reach of anyone willing to take the time to learn.

The creativity is the hard part.

What follows is an introduction to the electronics and programming parts of that equation. The creativity part is entirely up to you. The more you can think about how to apply these ideas, the better chance you have of finding that creative spark.

0.1 What to buy

The truth is this is an impossible question to ask because of several factors. 1. You will have your own ideas of what you're going to want to do, and that will drive your decisions. 2. The technology available changes constantly and at a breathtaking pace. 3. The programming languages that glue this all together also change, albeit less quickly.

So what follows is an attempt to suggest things that could cover a wide variety of applications, but have been specifically chosen for this book and the project described in later chapters.

If you are planning an introductory classroom based course: The first half of this book uses the Circuit Playground Express. It's an excellent, all-in-one beginner board that has a ton of features on it: lights, sensors, and can be programmed multiple ways. In a single board, you can program it with a visual, web-based interface from Microsoft called MakeCode that's largely aimed at kids and you can program it with Circuit Python. This is great for students who have no experience with code: You can use MakeCode to do drag-and-drop programming to show what you can do and then switch to code to make the transition from the real to the abstract a smoother ride. There is a Circuit Playground Express Educator's Pack that gets you 15 boards, 15 USB cables and some fun extras for, at the time of this writing, $350.

If you are planning a project: For project work, you are going to want more flexibility in a smaller form factor. For this book, we're going to use the Adafruit Feather series, specifically the Adafruit Feather Sense, which packs a sensor array similar to the Circuit Playground Express, and we're going to add the Adalogger FeatherWing, an add on to give us a real time clock and an microSD card reader for data logging. For that we get sensors, clock, data logging and external power management for $38 a board. Could you do this cheaper by buying separate sensors and building this yourself? Without question. But it would be difficult and time consuming for beginners. Time is money.

0.2 Why circuit python?

This is largely personal opinion, but there's a solid argument here.

1. Arduino has been the most visible small electronics platform for years and years. The one knock on it? It's based in C, which is an especially difficult language for beginners to learn.
2. Python is one of, if not the fastest growing programming language in the world right now. There's reasons beyond this project for students to learn Python.
3. Python is one of the more beginner friendly languages to learn. The syntax is largely straightforward and the fiddly parts are manageable with practice.

4. It is much easier to see what your code is doing in Python versus the old Arduino code.

The downsides to using Circuit Python, in the interest of fairness:

- Micro Python, and the Circuit Python variant, are relatively new.
- The main consequences of that newness are that there's years and years of code, message board and tutorial support for Arduino out there and there just isn't that for Circuit Python … yet.
- There's fewer options for boards that support Circuit/Micro Python.

On balance, I believe the benefit of students learning some Python outweighs the downsides. Especially if this is the only chance they'll have to learn a programming language. Connecting code to the physical world is a powerful way to learn it.

Chapter 1

Setup

The first step of any sensor journalism project is setting up your environment.

It is, unfortunately, a moving target. Fortunately, Adafruit keeps updated instructions available.

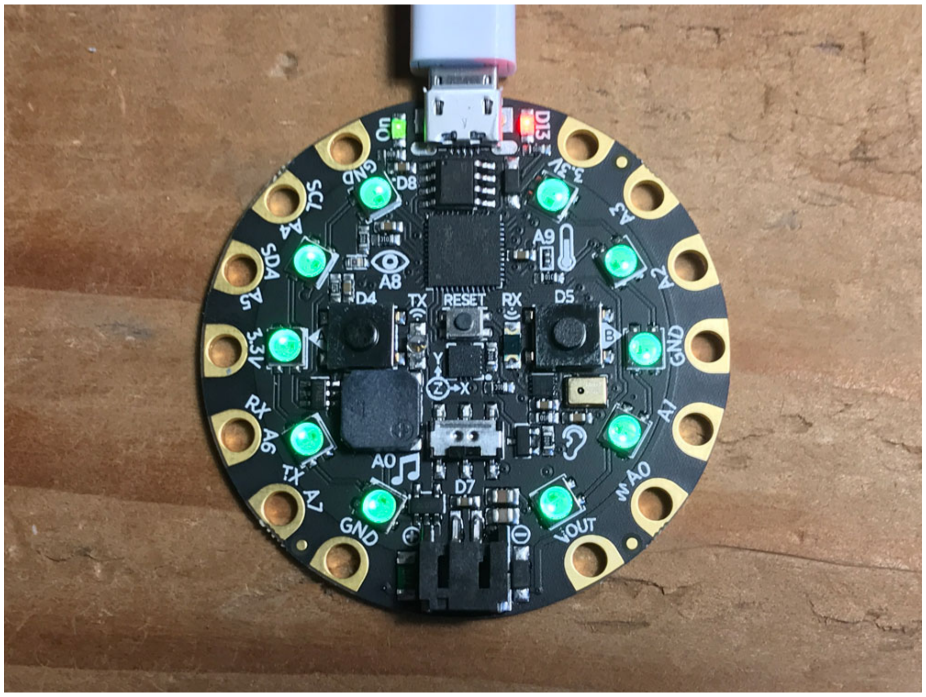
Windows Users: You will need to first install drivers.

All Users: You may to update the bootloader, especially if you are on a Mac.

1.1 First things first

Let's have some fun.

Plug in your Circuit Playground to a USB port on your computer. You should see two things happen: first, your LEDs should all turn green like this:



And you should now have a USB drive on your computer called CPLAYBOOT.

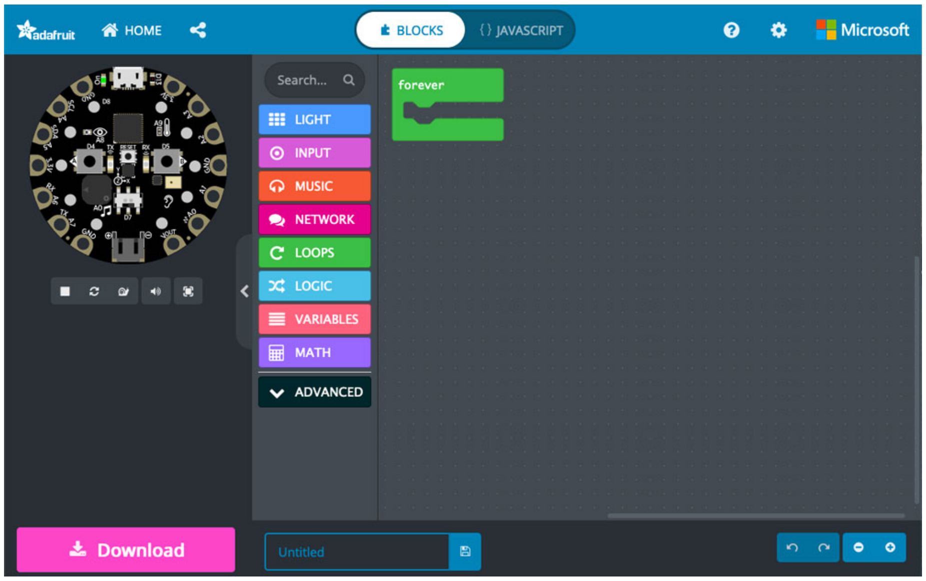
If that's the case, you're good to go. If not, review the Adafruit setup instructions.

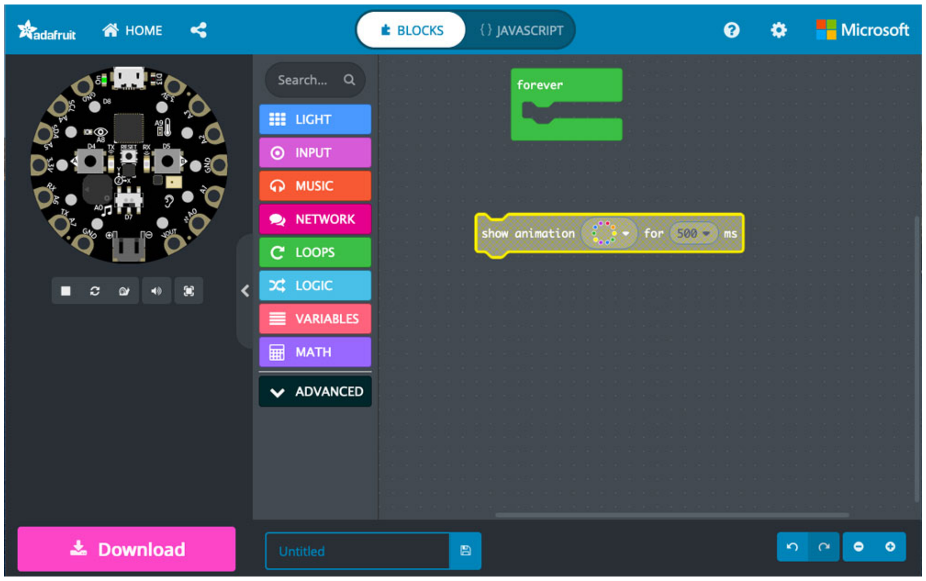
Let's go to the MakeCode site specifically for the Circuit Playground Express.

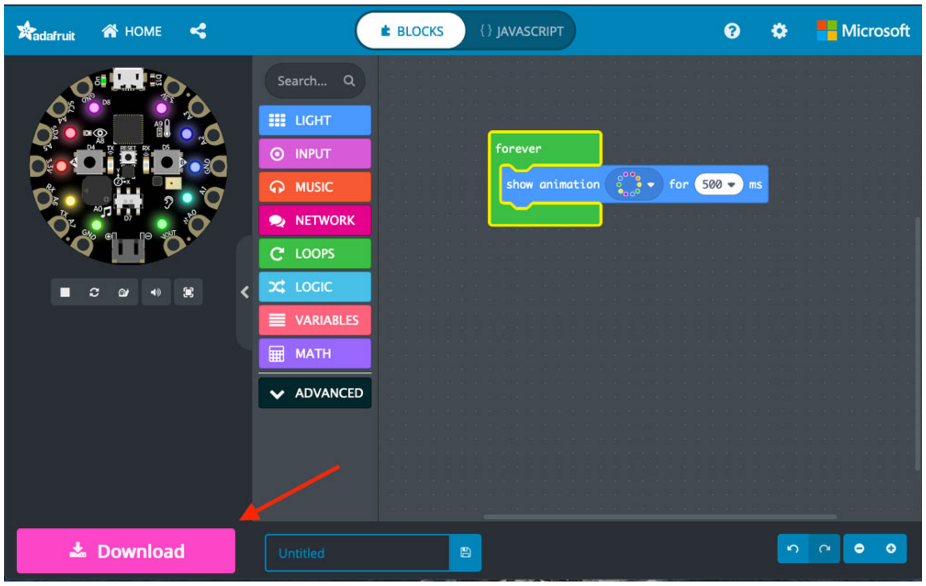
Click on the big blue New Project box.

You should get a workspace, with concepts you can work with on the left, and the execution space on the right, with a U shaped box that ominously says "forever" on it.

In this workspace, you can drag commands into the space, arrange them, and execute them on your board. To demonstrate, let's click on the Light box and drag out the "show animation for 500 ms" bar into the workspace. Then, drag it into the forever box so they snap together.


HOME
BLOCKS
{ } JAVASCRIPT
Microsoft
Search...
LIGHT
INPUT
MUSIC
NETWORK
LOOPS
LOGIC
VARIABLES
MATH
ADVANCED
forever
Download
Untitled

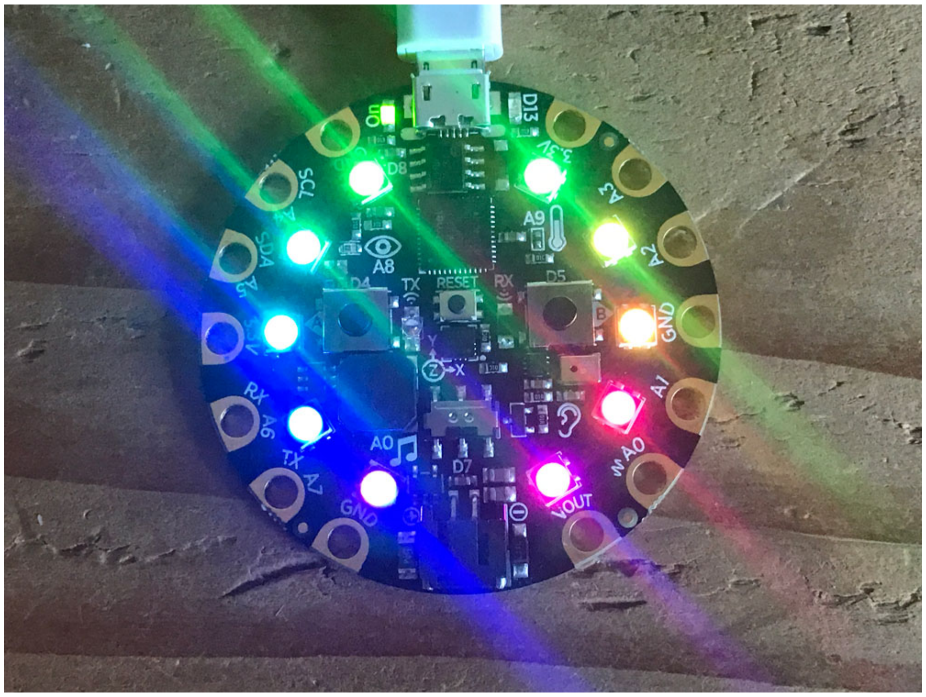

HOME
BLOCKS
{ } JAVASCRIPT
Microsoft
Search...
LIGHT
INPUT
MUSIC
NETWORK
LOOPS
LOGIC
VARIABLES
MATH
ADVANCED
forever
show animation for 500 ms
Download
Untitled



Then, click the Download button. The code to run on your board will download to your Downloads folder with a .uf2 extension on it. Drag that file into your CPLAYBOOT drive. When that happens, CPLAYBOOT is going to disappear (and on a Mac, you'll get that dreaded Disk Not Ejected Properly warning that you can, in this case, completely ignore).

What's your Circuit Playground Express doing? Is it spinning a rainbow at you over and over?

Congratulations, you're a hardware programmer.



1.2 Installing Circuit Python

Your Circuit Playground is capable of running both MakeCode and Circuit Python. To install Circuit Python, you'll need to follow a few steps:

1. Install the latest version of Circuit Python on your board.
2. Install mu-editor, a simple and free code editor for Circuit Python.

Once you've got that going, time to look at some Python basics.

Chapter 2

Python basics

Python is a great language to learn if you've never seen code before, or if you have little experience with it. With a little bit of practice, you can get to reading someone else's Python code and have a decent idea of what it is doing. There is a lot of code to borrow on the internet, and a fair number of times you can find what you need on a site or a message board.

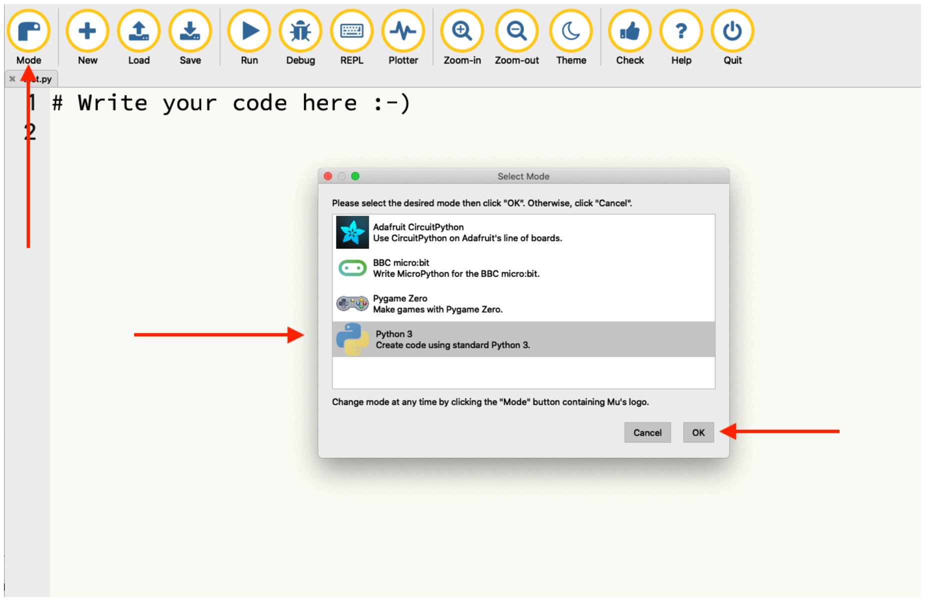
But the copy and pray method of development is not a good plan. Better to learn some basics and grow into some knowledge.

The basics that you need to get started with Python – and by extension, Circuit Python – are:

1. Variables
2. Functions
3. Loops
4. Conditionals
5. Error trapping
6. Libraries.

It's just six things, and we'll go through each one here. As we go forward with making your board do things, we'll highlight these concepts and how most of what we're doing is just mixing them around in new and creative ways.

In the last chapter, you installed mu-editor. Open mu-editor, then click on Mode and choose Python 3.

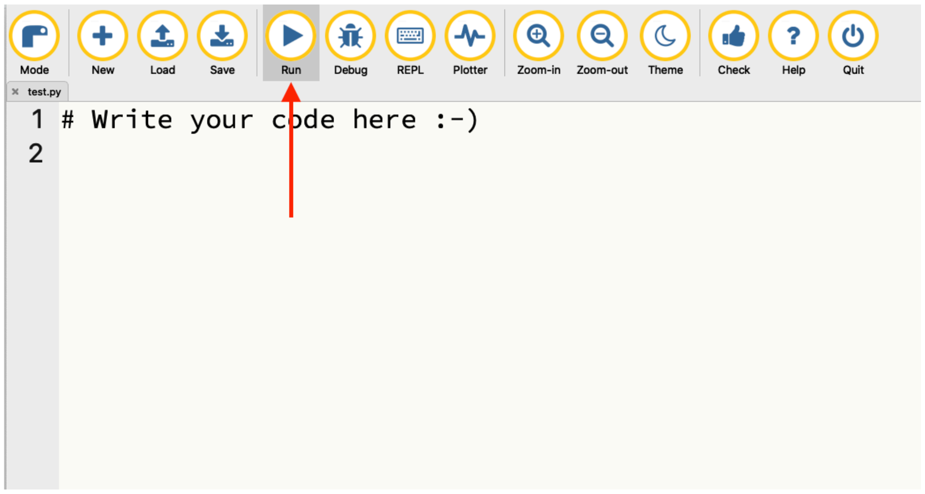

Mode
New
Load
Save
Run
Debug
REPL
Plotter
Zoom-in
Zoom-out
Theme
Check
Help
Quit
test.py
Select Mode
Adafruit CircuitPython
Use CircuitPython on Adafruit's line of boards.
BBC micro:bit
Write MicroPython for the BBC micro:bit.
Pygame Zero
Make games with Pygame Zero.
Python 3
Create code using standard Python 3.
Cancel
OK

```
# Write your code here :-)
```

Please select the desired mode then click "OK". Otherwise, click "Cancel".

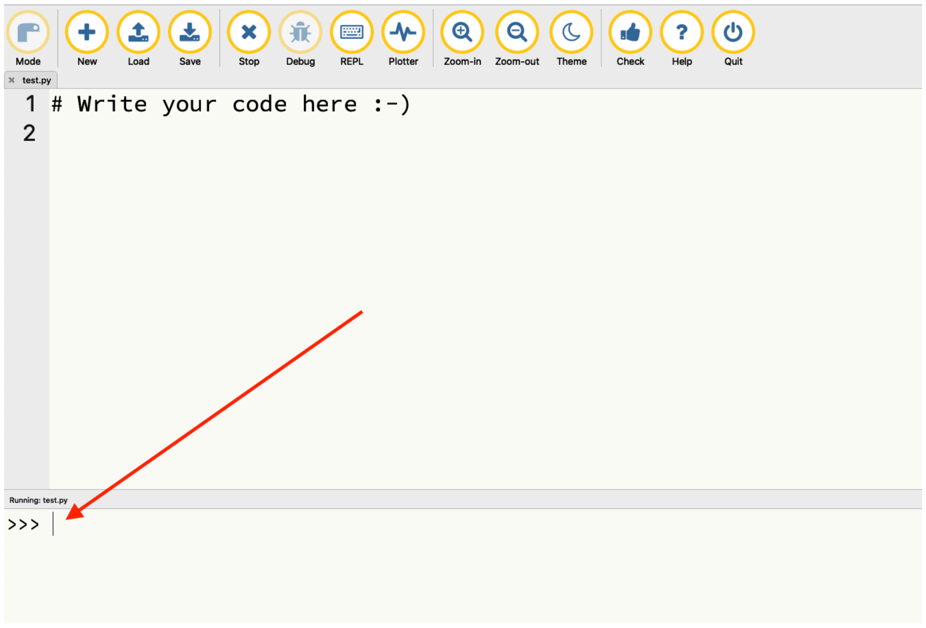
Change mode at any time by clicking the "Mode" button containing Mu's logo.

Then, just click the Run button.



```
# Write your code here :-)
```

That should pop up a pane at the bottom of your window like this:



```
# Write your code here :-)
```

```
>>>
```

In that pane, next to the `>>>` you'll be able to just type Python code and execute it. So let's try some.

2.1 Variables

One of your first lessons is to not make things more complicated than you need. If you've been told that programming is hard, the truth is it can be, but it doesn't have to be. And if you've been told that things are highly technical and you wouldn't understand them, that's a lie.

Most of programming is very simple. The complexity comes in what we want to do and how those things mix together. Fortunately for us, most of what we want to do is simple, using simple concepts and easy to understand ideas.

The variable is your first lesson in simple.

A variable is just a thing that stores stuff. That's it. It could be a number. Or some text. It could be a whole list of numbers, the entire text of War and Peace, or every reading from one of our sensors. It can be anything in our computers, or on someone else's in some cases.

Here's where variables come into play.

First, python can act like a calculator. We can just add two numbers together. Try this in your terminal.

```
2 + 2
```

```
## 4
```

Simple, right? We know the answer to this before we run it and, sure enough, Python gives us the right answer.

But what if we want to store the result of that math to use later? We use a variable. And in Python, variables are created with an equal sign. Type this in your terminal and hit enter.

```
number = 2 + 2
```

Uh oh. Seems like nothing happend, right? We typed something and we got nothing. Well, let this be another lesson – code does exactly what you tell it to do, not what you want it to do. In this case, we told it to store 2 + 2 into a variable called number. We didn't say show us the number. We didn't say anything else. Just store it.

Want to see it? Type this an hit enter:

```
print(number)
```

```
## 4
```

This does what you think it does. It prints out (to the screen) your variable. This is really a function, which we'll discuss in a bit, but any time you want to see the thing you're working with, just print it.

In Python, how things like variables are formatted matters. Notice our numbers are just numbers: `2`. A number, without any quotation marks, is automatically formatted as a number. To automatically format something as text, you put it in quotes. Type this:

```
print("I am learning Python")
```

```
## I am learning Python
```

What happens if you don't put it in quotes?

```
print(I am learning Python)
```

You get an invalid syntax error. That's because without the quotes, Python thinks I, am, learning and Python are variable names.

Where this gets hard is numbers that aren't numbers. We looked at something simple like `2+2`. But what happens when you have a zip code? Is a zip code a number? The answer is no – you would never do math with a zip code. So `2` is a number, `"2"` is text. What happens if you try to add text together?

```
"2" + "2"
```

```
## '22'
```

If your reaction is … uh, what? … there's something you need to know. In Python, `+` between two numbers means add, but `+` between two pieces of text means merge these two pieces of text together. So first – see that `"2"+"2"` equals `'22'`. But see the single quote marks around them? That means that 22 is really `"22"` or text, not a number.

2.2 Functions

You've already used a function when you printed something out. The definition of a function is a piece of code that takes input and returns output. That's it. Input. Output. Your `print()` function took in your variable and printed it out.

The basic rule of thumb on functions is that if you are going to do it twice, you should make it a function. Let's make a simple function to convert Celsius to Fahrenheit. To convert Celsius to Fahrenheit, you take your number, multiply it by 1.8 and then add 32. We can do this with a calculator easily. Some of you might be able to do it in your head. But what if you wanted to do it every 10 seconds? You want a function.

To create a function, you define it with `def`, give it a name, and then name your input value.

> **An important note about Python:** Python has what is called Meaningful White Space, meaning indentations matter. Code that is indented under code that isn't indented is considered part of that code – a continuation of it. Then, when the code returns to not being indented again, the block of code is considered done. **In Python, indentations are made with four spaces, NOT tabs.

Our function will create a variable called `fahren`, short for Fahrenheit, and then do the math. Our output will print the Fahrenheit value.

```
def convert(temp):
    fahren = (temp * 1.8) + 32 # note the four spaces
    print(fahren) # note the four spaces
```

Now, to use it, we can just call `convert` and give it a number. What is 28 Celsius in Fahrenheit?

```
convert(28)
```

```
## 82.4
```

So 28 Celsius is 82.4 Fahrenheit. A nice warm day.

Later, we'll explore functions that other people have made for us when we talked about external libraries. The important things to remember – functions take input and provide output.

2.3 Loops

Loops are the most powerful feature in programming. Just about every website and app that you use in a day are, at their core, a loop. Loops repeat a series of commands on a piece of data until either there is no more data to work with or, in some cases, forever. For example, let's use Instagram. When you log into Instagram, a request goes to Facebook's servers and gets the latest photos from your friends and the people you follow. Those photos are then put into a list. Your list might have a few new photos or it may have hundreds. We don't know until we look. Then, on your phone, those photos appear, one after another, until the list runs out (when another request is fired off, bringing you more photos in an attempt to keep you there longer).

That's a loop.

So first, let's create a list of Celsius readings. Lists in Python are created with brackets:

```
myreadings = [20.2, 20.8, 21.1, 20.9, 20.7]
```

One of the most beginner friendly parts of Python is the simplicity of loops. When you see it, you can pretty much read it right away. There's two kinds of loops. Let's start with the `for` loop. In English, what Python's `for` loop code is saying is this: For each instance of a thing in a list of things, do this. Here's what that looks like in code:

```
for reading in myreadings:
    print(reading)
```

```
## 20.2
## 20.8
## 21.1
## 20.9
## 20.7
```

The first parts of that code – reading – can be anything we want it to be.

```
for thereisnoplacelikenebraska in myreadings:
    print(thereisnoplacelikenebraska)
```

```
## 20.2
## 20.8
## 21.1
## 20.9
## 20.7
```

The second part – allofmyreadings – is our list that we created. The second part must exist somewhere before we run the loop. A good coding habit to get into? Name things what they are. Don't get cute. And don't create things difficult to spell. Don't name things single letters. Give it a real name. So the first example is good, the second example, while dear to my heart, is bad.

So now we can combine our function and our loop.

```
for reading in myreadings:
    convert(reading)
```

```
## 68.36
## 69.44
## 69.98
## 69.62
## 69.25999999999999
```

2.4 Conditionals

Conditionals are where you can make choices about things. If this is true, do this, otherwise, do that. This is particularly handy for classifying data as one thing or another based on a rule.

So let's pretend for a second that in our list of readings, we have a threshold. Let's say that if the temperature rises to 21 C or more, we call that Warm. If it's not 21 C, then it's Not Warm.

We can label those using conditionals and operators like > and <.

```
for reading in myreadings:
    if reading >= 21:
        print("Warm")
    else:
        print("Not Warm")
```

```
## Not Warm
## Not Warm
## Warm
## Not Warm
## Not Warm
```

The other kind of loop is the `while` loop. They are similar, in that they repeat commands, but they do it until a condition isn't true. To read them, they say while this condition is true, do this thing until it's not. If you don't set a condition that could be false, the loop will run until you stop the code by hand.

For computers, this can be trouble – an infinite loop can become a stuck process and consume resources you didn't mean to consume. In sensors, it can be exactly what we want – take readings until I pull the plug. For purposes of this tutorial, I'm not going to go into much depth here – we'll have a chance to do it much more when we get to coding the board. But here's what they look like.

```
while True:
    print("Hello")
```

And if I were to run that, it would print Hello over and over and over and over until I made it stop (which wouldn't be long, because it would get annoying quickly).

Combining concepts

We can do this exactly the same way by adding variables into it. See the difference here?

```
for reading in myreadings:
    if reading >= 21:
        label = "Warm"
    else:
        label = "Not Warm"
    print(label)
```

```
## Not Warm
## Not Warm
## Warm
## Not Warm
## Not Warm
```

Does the same thing, but uses a variable – `label` – to store the decision of the conditional. These are tiny steps toward creating complex code using very simple ideas.

2.5 Error trapping

One issue you will run into with sensors is when they don't work. Things go wrong. Some sensors are quite "noisy", which is a way of saying the readings can bounce all around. Sometimes they don't do what you expect them to do. So we have to build code that can tolerate problems.

We can do that with error trapping. Simply put, error trapping is like saying "Try this, and if it works, do this, else, do this." And because Python prizes readability, you create error tolerance by using `try`.

Let's create a new list of readings from our thermometer, but this time, one of them is going to be an error.

```
newreadings = [20.2, 20.8, "ERROR", 20.9, 20.7]
```

Python lists don't really care what it is in them. You can store numbers, text, even other lists in a list and Python will store it dutifully. The problem is when you loop through that list and expect them to be all one thing. For example, let's apply our Fahrenheit converter to each reading in our list.

```
for reading in newreadings:
    convert(reading)
```

```
## Error in py_call_impl(callable, dots$args, dots$keywords): TypeError: can't multiply sequence
##
## Detailed traceback:
##   File "<string>", line 2, in <module>
##   File "<string>", line 2, in convert
```

Python error messages aren't the easiest thing to read – they're orders of magnitude better than many languages, but that's a low bar to get over. But we can see there is a giant hint in the first word of the error message: TypeError. That leads me to believe there's a data type problem, which we know is true.

To trap that, we need to use `try` and `except`. Except, in this case, means what to do if it doesn't work.

```
for reading in newreadings:
    try: # try doing what we want done
        convert(reading)
    except: # here's what to do if it breaks
        continue # this means just skip it, move along
```

```
## 68.36
## 69.44
## 69.62
## 69.25999999999999
```

Now it works, but our choice of using `continue` means we get four readings, not five. Sometimes, that's an acceptable outcome. If it didn't work, we don't want it. We can do other things.

```
for reading in newreadings:
    try: # try doing what we want done
        convert(reading)
    except: # here's what to do if it breaks
        print("Ooops")
```

```
## 68.36
## 69.44
## Ooops
## 69.62
## 69.25999999999999
```

It's generally a bad idea to mix data types, so maybe "Ooops" isn't a good choice. Some data encodes missing data as an impossible number, something like 99999. If it's 99999 degrees F, we've got much bigger problems than our sensor not working correctly for a reading.

```python
for reading in newreadings:
    try: # try doing what we want done
        convert(reading)
    except: # here's what to do if it breaks
        print(99999)
```

```
## 68.36
## 69.44
## 99999
## 69.62
## 69.25999999999999
```

Then we know, in our data analysis, to filter those out or annotate those readings.

The point being, you have choices on how you handle errors. You need to think through what your application requires. Do you need to know what happens on each reading, regardless if it works or not? Or is a missing reading here and there not important? Think it through before you start coding.

2.6 Libraries

The strength of any programming language is the quality of available external libraries. Python has many, many contributed libraries to do a mind boggling number of tasks. Many of them are incorporated into something called the standard library – a common set of high quality libraries that aren't part of the base language but are so good that everyone gets them. Circuit Python has theirs, and we'll get into those soon, but using libraries is an important concept to understand.

Good news: It's pretty simple.

There's two ways to use an external library – you can import it whole, or you can import what you need.

Let's say we need an average temperature of our readings for our error problems. Instead of a blank or a fake number, we're going to replace that error with the average of all readings. In a controlled environment, where the values aren't going to change wildly unless there's a problem, this might be a good solution to a touchy sensor.

So we could do this with code we know. We can add up all the values in the list, then divide them by the number of items in the list. But we aren't the first people to need this. This is a common thing, so any time we run into something common like this, your first thought should be "I wonder if there's a library that does this."

Python has a library called `statistics` that has a `mean` function in it.

The import it whole way involves importing the library, then using dot notation to get to it. Think of dot notation like a line that connects a library to a function within it. It looks like this:

```
import statistics

statistics.mean(myreadings)
```

```
## 20.74
```

So the average temperature in my list of readings is 20.74.

Purists will tell you that importing the whole library for one function is inefficient and bad form. So we can import the one function and use it like this:

```
from statistics import mean

mean(myreadings)
```

```
## 20.74
```

Two different ways, same result.

We'll make use of external libraries when we start with Circuit Python. The important lesson here is that you can import a library and use functions within it.

2.7 Putting it all together

Let's combine some ideas. See if you can do this yourself.

1. We're going to loop through the list of temperature readings below.
2. We're going to convert them from Celsius to Fahrenheit using our function.
3. If that doesn't work, we'll substitute the average temperature using try/except.
4. Then, we'll print it out to the screen.

```
testreadings = [20.3, 20.6, 20.1, 21.1, 21, "ERROR", 20.9, "ERROR", 21.2]
```

First things first, what happens if we try to get an average of that list?

```
mean(testreadings)
```

```
## Error in py_call_impl(callable, dots$args, dots$keywords): TypeError: don't know how to coerce
##
## Detailed traceback:
##   File "<string>", line 1, in <module>
##   File "/Users/mwaite3/Library/r-miniconda/envs/r-reticulate/lib/python3.6/statistics.py", lin
##     T, total, count = _sum(data)
##   File "/Users/mwaite3/Library/r-miniconda/envs/r-reticulate/lib/python3.6/statistics.py", lin
##     T = _coerce(T, typ)  # or raise TypeError
##   File "/Users/mwaite3/Library/r-miniconda/envs/r-reticulate/lib/python3.6/statisti
##     raise TypeError(msg % (T.__name__, S.__name__))
```

Uh oh. TypeError again.

So we need to remove non-numbers from our list.

There's a lot of ways to do this, but one way is to use a loop and a try/except. We'll create a new list, called `cleanreadings` and we'll use a new bit of code – `append` – to add things to that new list. Then, we loop.

In the try/except, we'll try to do something that will only work with numbers – like convert it to a floating point number or a number with a decimal point. If that succeeds, then add the number to the list. Otherwise, move along.

Then we can calculate the mean and save it as a variable.

```python
cleanreadings = []

for reading in testreadings:
    try:
        float(reading)
        cleanreadings.append(reading)
    except:
        continue
```

```
## 20.3
## 20.6
## 20.1
## 21.1
## 21.0
## 20.9
## 21.2
```

```python
avgreading = mean(cleanreadings)
```

Now we can do our conversion.

```python
for reading in testreadings:
    try:
        convert(reading)
    except:
        convert(avgreading)
```

```
## 68.53999999999999
## 69.08000000000001
## 68.18
## 69.98
## 69.80000000000001
## 69.33714285714285
## 69.62
## 69.33714285714285
## 70.16
```

Chapter 3

Circuit Python

Now it's time to start making your board do something.

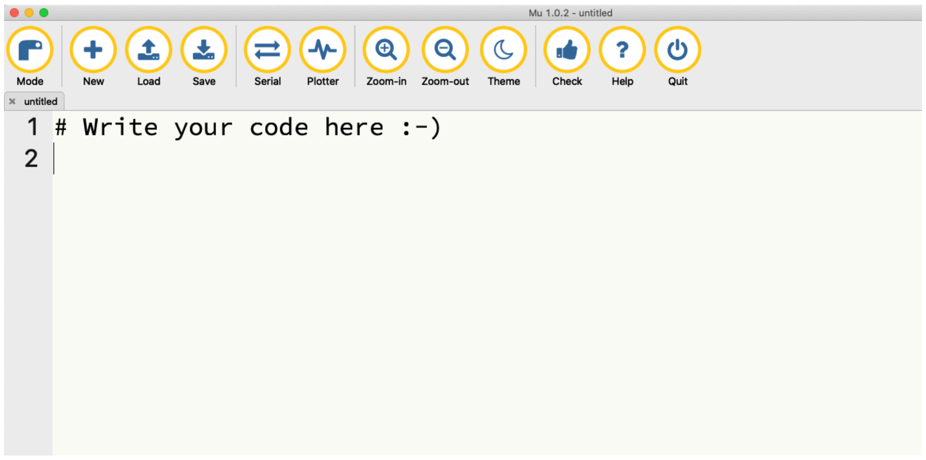
If you've still got mu-editor running your test.py file, you can click the Stop button. Then click the X on the test.py tab and close it.

Now, plug your Circuit Playground into a USB port on your computer. If you've been following along, it should have a drive on your computer called CIRCUITPY. If not, here's the instructions for installing Circuit Python.

An alert box should appear that says it detected your board.

In mu-editor, click Mode and select Adafruit CircuitPython.

Then click New. You should have a window that looks like this.



```
# Write your code here :-)
```

The first thing you'll notice is that it says `# Write your code here :-)`. That's what's called a comment. Comments start with a `#` and everything after it on that line will not be executed as code. So you can write whatever you want. You can also delete that line. Or keep it if you need the reminder. Either way, it's not going to do very much.

The first thing we're going to do to get started is we're going to make a tiny light blink. It's a good place to start because we're going to learn:

1. How to write code in mu-editor.
2. How to save code in mu-editor.
3. How to move that code onto your board.
4. How to make a light blink.

And in that order. Making a light blink isn't all that important, but the lessons learned doing it will be useful throughout.

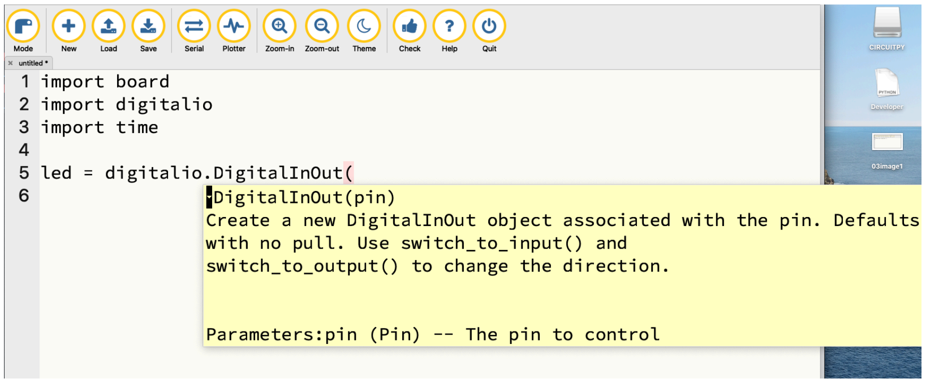
This is the code we're going to use. DON'T JUST COPY AND PASTE IT. I want you to see a feature of mu-editor and other code editors. It's got many names, but it's aware of what you're typing and is going to try to make useful suggestions and offer up some simple documentation.

```
import board
import digitalio
import time

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

while True:
    led.value = True
    time.sleep(.5)
    led.value = False
    time.sleep(.5)
```

As you type, you should see suggestions like this:



```
import board
import digitalio
import time

led = digitalio.DigitalInOut(
```

```
DigitalInOut(pin)
Create a new DigitalInOut object associated with the pin. Defaults
with no pull. Use switch_to_input() and
switch_to_output() to change the direction.


Parameters:pin (Pin) -- The pin to control
```

Let's walk through the code step by step.

First, the imports:

```
import board
import digitalio
import time
```

In the previous chapter, we talked about external libraries that do common things. Here's where you start to see how deep that hole can go. These libraries, in order, give you access to the pieces and parts on your board, handles digital input and output to and from your board and gives you some tools to handle time.

The best practice for your code is to do your imports first, at the top.

Now, we do some setup.

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

The first thing we're going to do, is set up a variable called `led`. What `led` becomes is a Python thing called a Class. A Class is a generic description of a thing. In this case, the Class covers Digital Input and Output tools on your board. They're called pins. In this case, `led` is going to control digital pin 13 – which we get from our `board` library. The D in front of the 13 means "digital". So `led` is going to control the input and output coming from digital pin 13.

The second thing is we need to set the direction of that input and output. This is easy when you're dealing with an led. It's a light. It's only output. So `led.direction` sets the direction of input or output to D13 to "OUTPUT".

Now, to the actual parts that make the board do things:

```
while True:
    led.value = True
    time.sleep(.5)
    led.value = False
    time.sleep(.5)
```

This says `while True` – without saying what is true or how to falsify it, so it's saying while the board is on – set the led value to True, or on. Then, pause a half a second. Then turn it off. And pause half a second. Rinse and repeat forever.

Save this file and call it code.py

It has to be called code.py for it to work. When you load it onto the board, it will be looking for code.py.

Now take that code.py file and drag it into the CIRCUITPY drive.

Is a little red light blinking?

Now we can play with it a little.

Change the values in time.sleep. Save the file. Drag it over to your board. What happens?

3.1 Making it easier

How we just accomplished that is what could be described as the long way around doing this. We went to the very specific pieces and interacted with them. This will be an important pattern to rely on later, when we get to more complicated problems.

But in this case, it can be easier.

Adafruit has a library specific to their board, which connects things much more directly. Instead of having to connect to digitalIO and set things and name the specific pin on the board, Adafruit has made it easier by just creating a thing called red_led.

```
import time
from adafruit_circuitplayground import cp

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

So this imports a thing called cp, which … you can guess what that stands for … and pretty much uses the while code from before to turn the red_led on and off.

We can make it even easier with a little logic game.

```
import time
from adafruit_circuitplayground import cp

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

But wait, how does this work? It works by exploiting what `cp.red_led` can be. It can either be `True` or it can be `False`. The `not` in this case says set cp.red_led to the other thing – if it's `True`, make it `False` – then pause a half a second. Then, if it's `False`, make it `True`, and pause. Repeat forever.

3.2 More fun with lights

Now that we've been introduced to Adafruit's board library, we can start to play with more things. Note there's much bigger lights than that tiny red led.

Let's play with those.

Those lights are called NeoPixels, or in the Adafruit library, `pixels` for short. And we can `fill` those with whatever color we want.

How about red?

```
from adafruit_circuitplayground import cp

while True:
    cp.pixels.fill((255, 0, 0))
```

Or green?

```
from adafruit_circuitplayground import cp

while True:
    cp.pixels.fill((0, 255, 0))
```

Or blue?

```
from adafruit_circuitplayground import cp

while True:
    cp.pixels.fill((0, 0, 255))
```

You can make them blink.

```
import time
from adafruit_circuitplayground import cp

while True:
    cp.pixels.fill((0, 0, 255))
    time.sleep(.5)
    cp.pixels.fill((0, 0, 0))
    time.sleep(.5)
```

You can light up just one of them.

```
from adafruit_circuitplayground import cp

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

Now, to understand how this works, you need to understand something about Python. Python counts in a weird way to everyone except programmers. Python starts counting at 0. So cp.pixels[0] is saying go to the first pixel and turn it blue. The rest are off.

3.3 Stretch your head

Before you run this, what do you think it's going to do?

```
import time
from adafruit_circuitplayground import cp

while True:
    cp.pixels[0] = (255, 0, 255)
    time.sleep(.1)
    cp.pixels[1] = (255, 0, 255)
    time.sleep(.1)
    cp.pixels[2] = (255, 0, 255)
    time.sleep(.1)
    cp.pixels[3] = (255, 0, 255)
    time.sleep(.1)
    cp.pixels[4] = (255, 0, 255)
    time.sleep(.1)
    cp.pixels[5] = (255, 0, 255)
    time.sleep(.1)
    cp.pixels[6] = (255, 0, 255)
    time.sleep(.1)
    cp.pixels[7] = (255, 0, 255)
    time.sleep(.1)
    cp.pixels[8] = (255, 0, 255)
    time.sleep(.1)
    cp.pixels[9] = (255, 0, 255)
    time.sleep(.1)
    cp.pixels.fill((0, 0, 0))
    time.sleep(.1)
    cp.pixels.fill((255, 0, 255))
    time.sleep(.1)
    cp.pixels.fill((0, 0, 0))
    time.sleep(.1)
    cp.pixels.fill((255, 0, 255))
    time.sleep(.1)
    cp.pixels.fill((0, 0, 0))
    time.sleep(.1)
    cp.pixels.fill((255, 0, 255))
    time.sleep(.1)
    cp.pixels.fill((0, 0, 0))
    time.sleep(.1)
```

The RGB value of 255, 0, 255 is full red and full blue – Purple. So it turns on the first, turns on the second, turns on the third and so on until all of them are lit up, turns them all off and on three times and repeats. All I did to make this?

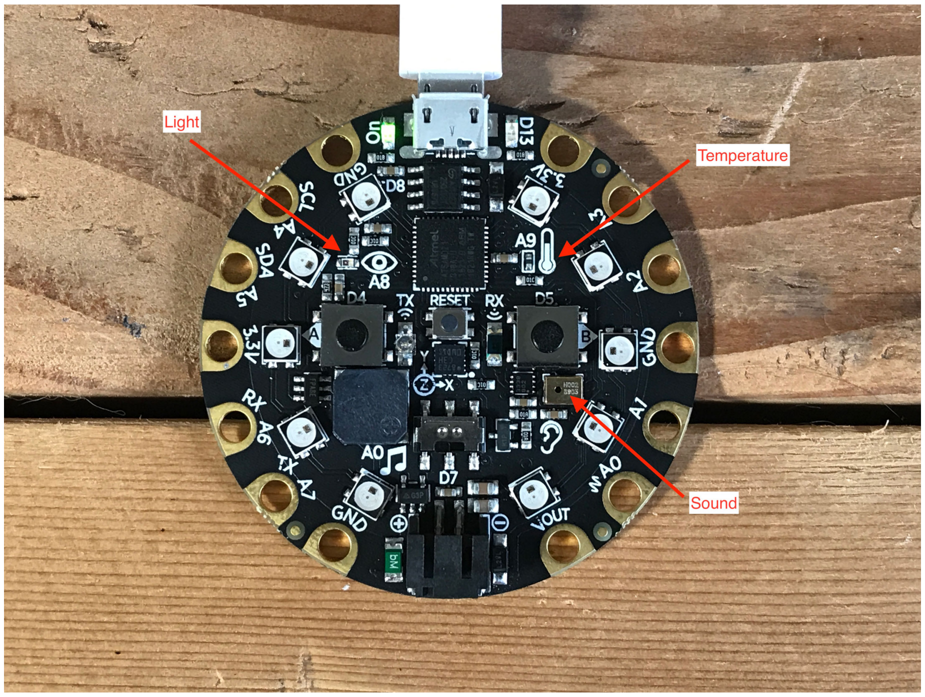
Copy, paste, change a value and repeat.

Chapter 4

Light sensing and what numbers mean

Up to now, we've been building toward sensing things. We've learned a little Python, wrote some code and make some lights turn on and off. But journalism isn't about blinking lights, it's about learning things about your environment and reporting them to others.

On your board are three sensors we're going to use in the next three chapters: The light sensor, the temperature sensor and the sound sensor. If you look at your Circuit Playground, you can find them by their symbols.



We'll start with light. Maybe your story needs to know how bright or dark it is in a place for periods of time. The light sensor on board can tell you on a continuous scale what level of light there is. According to Adafruit's documentation, the numbers range from from 0 – complete darkness – to 1023. Since the light sensor is an analog sensor, those are voltage measures.

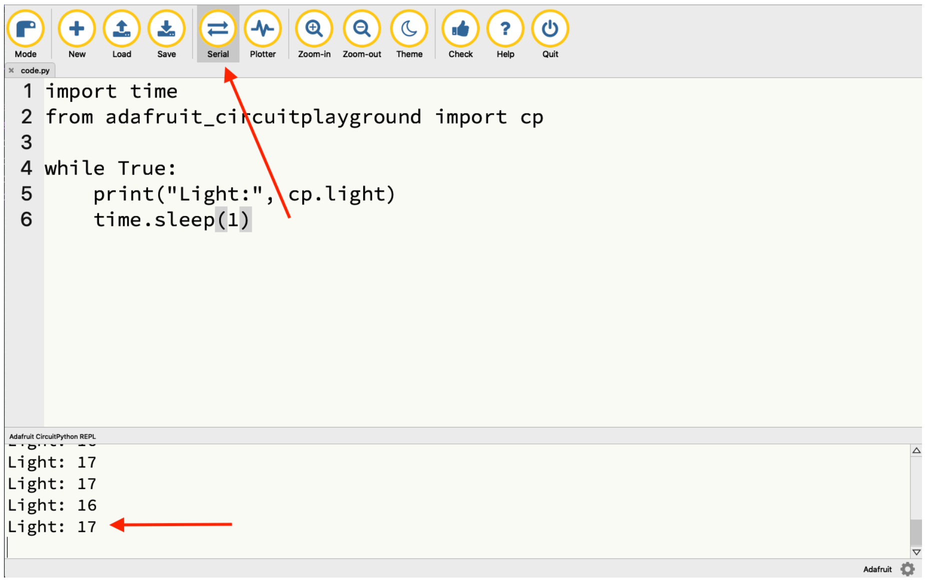
Now, here's where sensor journalism starts to get a little more difficult.

First, let's get some code going to illustrate this.

```
import time
from adafruit_circuitplayground import cp

while True:
    print("Light:", cp.light)
    time.sleep(1)
```

Save that to your board, and then click on the Serial button in the toolbar.



```
import time
from adafruit_circuitplayground import cp

while True:
    print("Light:", cp.light)
    time.sleep(1)
```

```
Light: 17
Light: 17
Light: 16
Light: 17
```

So you can see, my office is a little bit dim. I'm in a basement office with a half window next to me, but that window is partially underneath a deck, so the amount of sun coming in is indirect. So my light reading is 17. Now watch what happens when I bring in a lamp with a 100 watt equivalent LED in it.

With that very bright light – so bright you can't look at it directly without pain – the number goes up to 320.

I walked outside on a sunny summer day without a cloud in the sky and it topped out at 321.

But what does that mean? What does 321 mean? And if direct bright light is 321, what is 1023?

Light is measured in Lux, but would you know the difference between 1500 and 10000 Lux?

As journalists, we love numbers, but sometimes the number isn't that important. Maybe what's better here is to know that 320 is very bright and less than 50 is dim and maybe we come up with some categories in between. Ask yourself: do you need an exact number? Is that number meaningful? Or is it enough to just know the difference between very bright, bright, dim and dark?

4.1 Not to feed your love of numbers, but ...

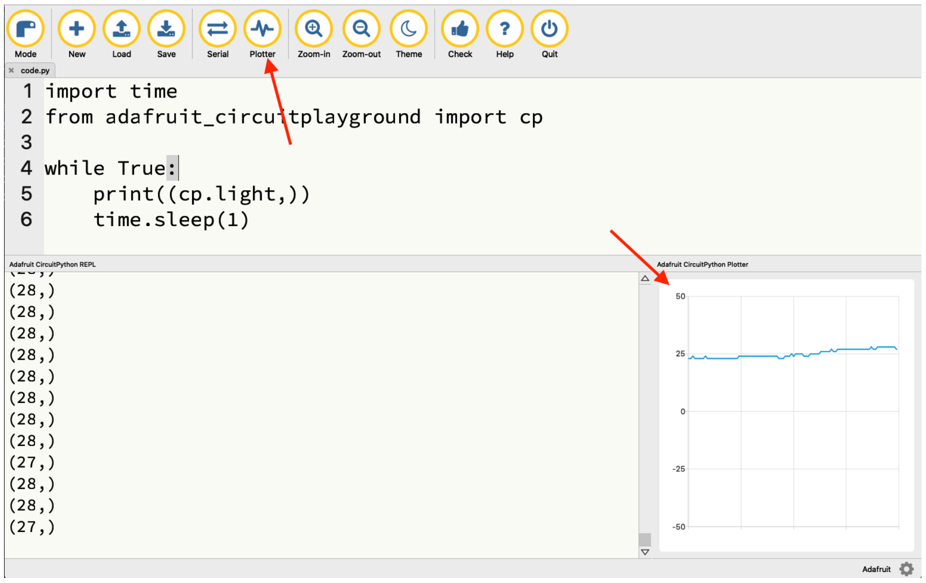
You may have noticed, when you clicked the Serial button, that's there's a Plotter button.

To use the plotter, we have to have something in Python called a tuple. A tuple is a kind of data that has two parts. You create them with parenthesis, and you separate the two elements of data in the tuple with a comma. To use the plotter, you're code has to produce a tuple. Here's one way.

```python
import time
from adafruit_circuitplayground import cp

while True:
    print((cp.light,))
    time.sleep(1)
```

Save that to your board and click the Plotter button. You should now get a line chart showing change over time.



```python
import time
from adafruit_circuitplayground import cp

while True:
    print((cp.light,))
    time.sleep(1)
```

```
(28,)
(28,)
(28,)
(28,)
(28,)
(28,)
(28,)
(28,)
(27,)
(28,)
(28,)
(27,)
```

And while that's cool to see, what does 27 mean? Something to think deeply about before you go forward with your sensor journalism project.

Chapter 5

Temperature and accuracy

Another sensor on the board is the temperature sensor. Unlike the light sensor, the data it returns is very easy to understand: It's the temperature in Celcius.

```
import time
from adafruit_circuitplayground import cp

while True:
    print(cp.temperature)
    time.sleep(1)
```

If you save that to your board and look at the Serial output, you'll see numbers probably around 24. That's 24 degrees Celcius. You can add things to the print statement to help you out.

```
import time
from adafruit_circuitplayground import cp

while True:
    print(cp.temperature, "C")
    time.sleep(1)
```

Remember in Chapter 2 when we converted C to F? Let's bring that back.

```
import time
from adafruit_circuitplayground import cp

def convert(temp):
    fahren = (temp * 1.8) + 32
    print(fahren)

while True:
    convert(cp.temperature)
    time.sleep(1)
```

Now that is telling me that my office is about 77.8 degrees Fahrenheit. Now I'm notoriously Dad Energy on this – air conditioning is expensive, so I don't care if you're warm – but my basement office is quite cool. It isn't 77.8.

So what is it?

That raises questions about how accurate are your sensors. The truth is this is one place where you get what you pay for. Do you need extremely accurate temperature readings because any change will cost your business money? Then you spend whatever it takes to get the most accurate sensor. Are you making a journalism project where a little fuzziness is not a huge deal? Then the sensor isn't going to cost as much.

But can we make this better without spending more money? We can.

5.1 Turning down the noise

When planning a sensor journalism project, there's several questions you're going to have to answer given your requirements and given your available equipment.

Two critical questions will be:

1. How accurate does the measurement have to be? Is good enough good enough?
2. How precise does the measurement need to be?

For example, my readings are saying things like 78.6844. Do we really need four digits of precision? Can you tell the difference between 78.6844 and 78.6855? Do people talk about temperature with decimals?

And notice that the sensor is sensitive enough that the temperature goes up and down with each reading, no matter how frequently you ask for it. We've got it set for every second. Do micro differences in temperature every second matter?

There are some things we can do in code to deal with these questions.

First, we can average a collection of temperature readings to smooth it out. To do this, we're going to create an empty list, then populate it with readings every second. Then, when we hit a threshold level – we'll start with 5 – we'll average them together. We'll clear out the list, and start it all over again.

```
import time
from adafruit_circuitplayground import cp

temps = [] # here's the empty list
```

```
def convert(temp):
    fahren = (temp * 1.8) + 32
    return fahren

while True: # do this forever
    while len(temps) != 5: # while the number of readings in our list isn't 5
        temps.append(convert(cp.temperature)) # append each temperature reading in Fahrenheit to
        time.sleep(1) # wait a second and repeat it again
    print(sum(temps)/5) # once the while loop breaks, we'll add them up and divide by 5 and prin
    temps.clear() # empty the list so the inner while loop can start again
```

That, by itself, smooths out the readings some.

But what else can we do?

Let's eliminate the decimal points.

```
import time
from adafruit_circuitplayground import cp

temps = []

def convert(temp):
    fahren = (temp * 1.8) + 32
    return fahren

while True:
    while len(temps) != 5:
        temps.append(convert(cp.temperature))
        time.sleep(1)
    avg = sum(temps)/5
    print("%.0f" % round(avg, 0)) # the first formats the output to have zero digits, the second
    temps.clear()
```

Now my sensor says its between 78 and 79 in my office, which is much more like how we are accustomed to talking about temperature. But it is right?

5.2 Low cost validation and calibration

As I said earlier, I'm pretty Dad Energy about the temperature in my house. But a basement office is generally cooler than the rest of the house – so much so that I have the vents closed so as to not add more cold air in the room. So how is it telling me that my office is 79 when the thermostat upstairs is set at 76?

One thing we can do is compare our sensor to an external sensor and compare. How different is our measurement from a better sensor.

My house has an Ecobee Smart Thermostat system and I've got a movable remote temperature sensor. Since I trust it to cool my house, I'm going to trust it here to give a better reading on the temperature in my office.

So what does the sensor say it is? 75.

The sensor, averaging five readings and averaging them together before rounding them to the nearest whole degree – now reads 78.

Simple solution? Knock three off our sensor reading to make it closer to the external sensor, which much more conforms with my understanding of what it actually feels like in my office.

Your situation is going to be different, so you'll need an external temperature gauge to validate your sensor output and adjust accordingly. But here's my code.

```
import time
from adafruit_circuitplayground import cp

temps = []

def convert(temp):
    fahren = (temp * 1.8) + 32
    return fahren

while True:
    while len(temps) != 5:
        temps.append(convert(cp.temperature))
        time.sleep(1)
    avg = (sum(temps)/5) - 3 # the - 3 is the adjustment to be as close to reality as
    print("%.0f" % round(avg, 0))
    temps.clear()
```

Chapter 6

Sound

```python
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,
    sample_rate=16000,
    bit_depth=16
)
samples = array.array('H', [0] * 160)
```

```python
mic.record(samples, len(samples))

while True:
    mic.record(samples, len(samples))
    magnitude = normalized_rms(samples)
    print(((magnitude),))
    time.sleep(1)
```

Chapter 7

Counting with buttons

```python
import time
from adafruit_circuitplayground import cp

count = 0

while True:
    if cp.button_b:
        count += 1
        cp.pixels[9] = (0, 255, 0)
        time.sleep(.2)
        print(count)
        cp.pixels[9] = (0, 0, 0)
    if cp.button_a:
        count -= 1
        cp.pixels[0] = (255, 0, 0)
        time.sleep(.2)
        print(count)
        cp.pixels[0] = (0, 0, 0)
```

Chapter 8

Touch