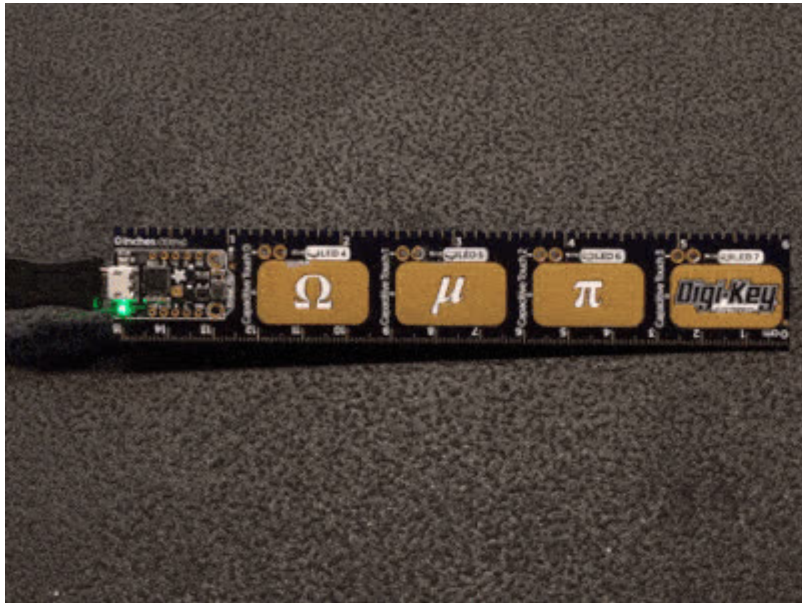
Simon Game for PyRuler and CircuitPython

Created by Isaac Wellish



https://learn.adafruit.com/simon-game-with-pyruler-and-circuitpython

Last updated on 2024-07-08 11:45:15 AM EDT

Table of Contents

Overview 3

- Reference guide
- Parts

CircuitPython 4

- Set up CircuitPython Quick Start!
- Further Information

Installing the Mu Editor 7

- Download and Install Mu
- Starting Up Mu
- Using Mu

Creating and Editing Code 9

- Creating Code
- Editing Code
- Back to Editing Code...
- Naming Your Program File

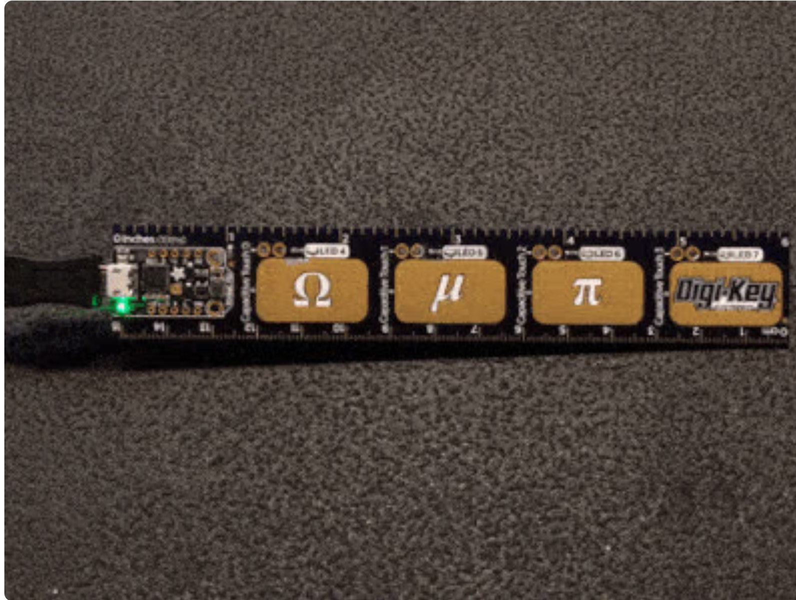
Code PyRuler with CircuitPython 14

- Installing Project Code

How Simon Works 17

- Game play
- How the code works
- Initialization
- Functions
- DotStar code
- Detecting capacitive touch and setting a timer
- Playing and reading each sequence with leds
- The main loop

Overview



Simon Says..."turn that ruler into an interactive memorization game with leds and touch pads!"

In this guide you can turn the Adafruit PyRuler into a clone of the classic game known as Simon (https://adafru.it/sTa)

Using CircuitPython, and the PyRuler's leds and capacitive touch pads we can make the PyRuler into our own Simon game. We can program the PyRuler to output patterns to the leds to memorize and then read capacitive touch inputs

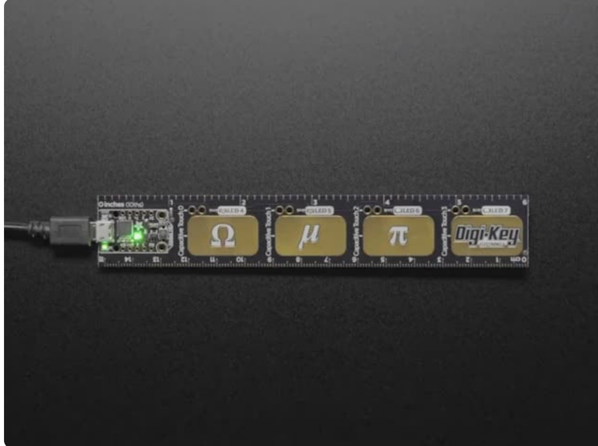
This guide and code was adapted from Miguel Grinberg's guide: Simon Game Clone with Circuit Playground Express and CircuitPython (https://adafru.it/Fym)

Reference guide

This is a great guide to use as a reference for all things PyRuler including pinouts, specs and more.

- Adafruit PyRuler (https://adafru.it/Fyn)

Parts



Adafruit PyRuler - Engineer Reference Ruler with CircuitPython

The first time you soldered up a surface mount component you may have been surprised "these are really small parts!" and there's a dozen of different names...

https://www.adafruit.com/product/4319

USB cable - USB A to Micro-B

This here is your standard A to micro-B USB cable, for USB 1.1 or 2.0. Perfect for connecting a PC to your Metro, Feather, Raspberry Pi or other dev-board or...

https://www.adafruit.com/product/592

CircuitPython

CircuitPython (https://adafru.it/tB7) is a derivative of MicroPython (https://adafru.it/BeZ) designed to simplify experimentation and education on low-cost microcontrollers. It makes it easier than ever to get prototyping by requiring no upfront desktop software downloads. The trinket M0 is the second board that comes pre-loaded with CircuitPython. Simply copy and edit files on the **CIRCUITPY** drive to iterate.

Your PyRuler already comes with CircuitPython but maybe there's a new version, or you overwrote your Trinket M0 with Arduino code! In that case, see the below for how to reinstall or update CircuitPython. Otherwise you can skip this and go straight to the next page!

If you have already plugged in your board, start by ejecting or "safely remove" the CIRCUITPY drive. This is a good practice to get into. Always eject before unplugging or resetting your board!

Set up CircuitPython Quick Start!

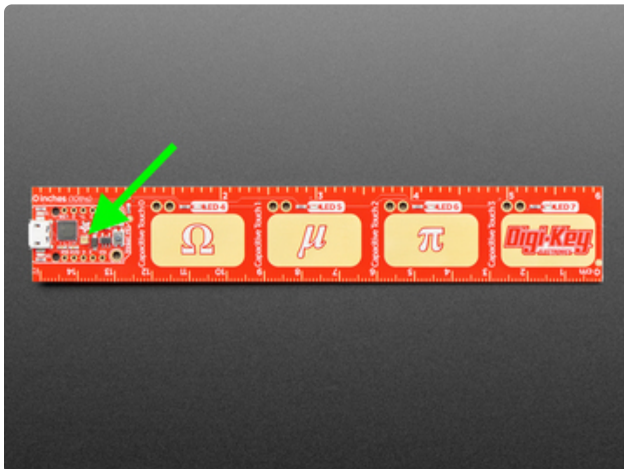
Follow this quick step-by-step for super-fast Python power :)

Download the latest version of CircuitPython for this board via circuitpython.org

https://adafru.it/Fst

Click the link above and download the latest UF2 file.

Download and save it to your desktop (or wherever is handy).

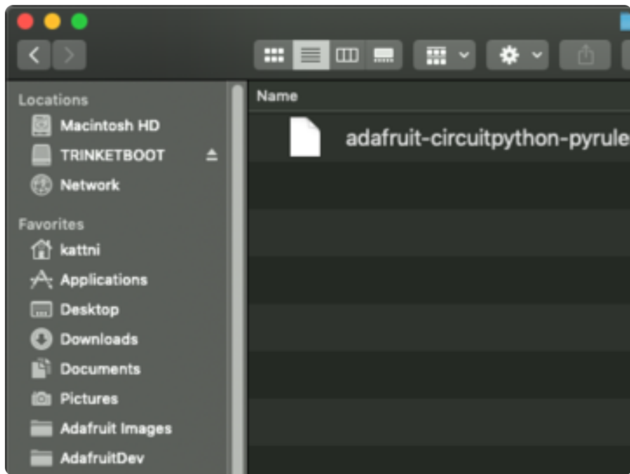


Plug your PyRuler into your computer using a known-good USB cable.

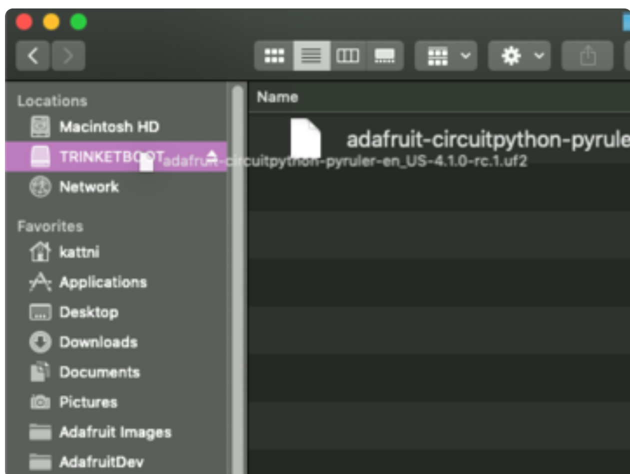
A lot of people end up using charge-only USB cables and it is very frustrating! So make sure you have a USB cable you know is good for data sync.

Double-click the small **Reset** button next to the Trinket M0 name printed on your board, and you will see the Dotstar RGB LED, noted by the green arrow in the image, turn green. If it turns red, check the USB cable, try another USB port, etc. **Note:** The little LED above the USB connector will be red - this is ok!

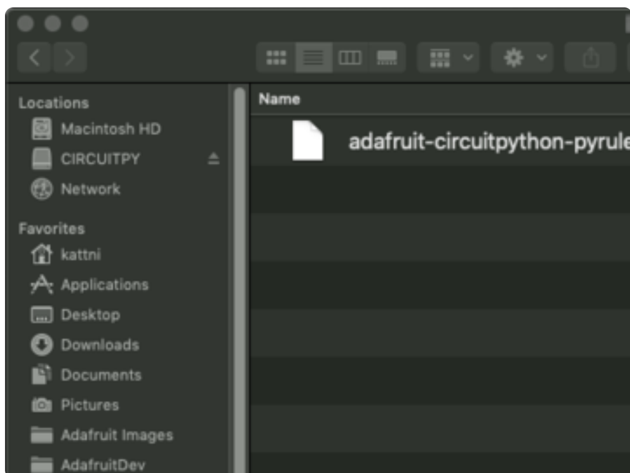
If double-clicking doesn't work the first time, try again. Sometimes it can take a few tries to get the rhythm right!



You will see a new disk drive appear called **TRINKETBOOT**.



Drag the **adafruit_circuitpython_etc.uf2** file to **TRINKETBOOT**



The red LED will flash. Then, the **TRINKETBOOT** drive will disappear and a new disk drive called **CIRCUITPY** will appear.

That's it, you're done! :)

Further Information

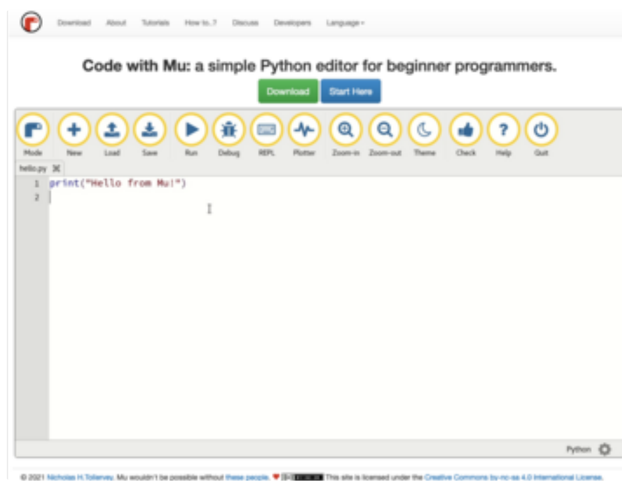
For more detailed info on installing CircuitPython, check out Installing CircuitPython (https://adafru.it/Amd).

Installing the Mu Editor

Mu is a simple code editor that works with the Adafruit CircuitPython boards. It's written in Python and works on Windows, MacOS, Linux and Raspberry Pi. The serial console is built right in so you get immediate feedback from your board's serial output!

Mu is our recommended editor - please use it (unless you are an experienced coder with a favorite editor already!).

Download and Install Mu



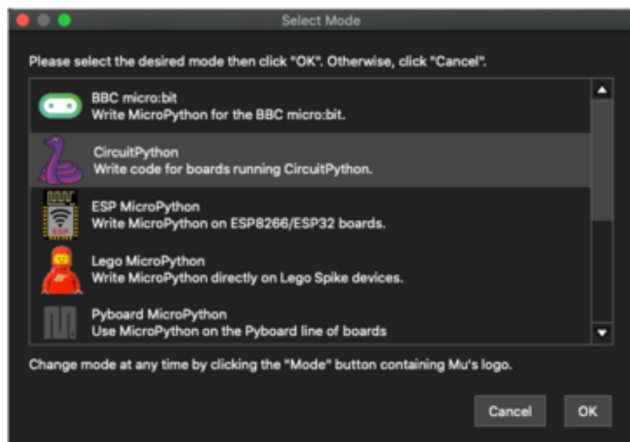
Download Mu from https://codewith.mu (https://adafru.it/Be6).

Click the **Download** link for downloads and installation instructions.

Click **Start Here** to find a wealth of other information, including extensive tutorials and and how-to's.

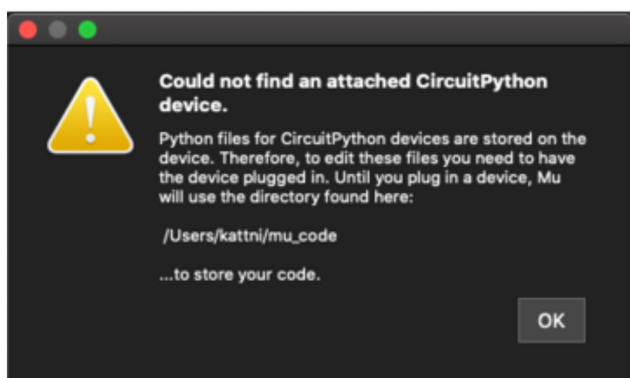
Windows users: due to the nature of MSI installers, please remove old versions of Mu before installing the latest version.

Starting Up Mu



The first time you start Mu, you will be prompted to select your 'mode' - you can always change your mind later. For now please select **CircuitPython**!

The current mode is displayed in the lower right corner of the window, next to the "gear" icon. If the mode says "Microbit" or something else, click the **Mode** button in the upper left, and then choose "CircuitPython" in the dialog box that appears.

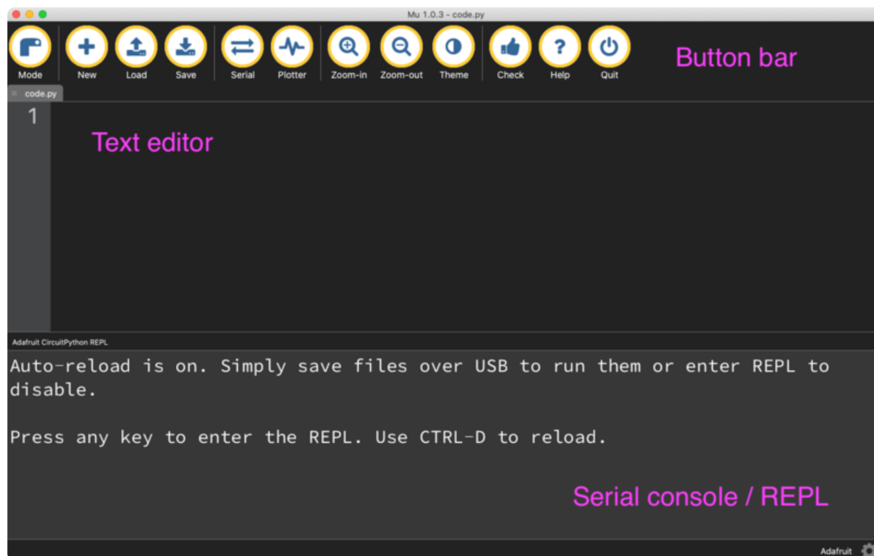


Mu attempts to auto-detect your board on startup, so if you do not have a CircuitPython board plugged in with a **CIRCUITPY** drive available, Mu will inform you where it will store any code you save until you plug in a board.

To avoid this warning, plug in a board and ensure that the **CIRCUITPY** drive is mounted before starting Mu.

Using Mu

You can now explore Mu! The three main sections of the window are labeled below; the button bar, the text editor, and the serial console / REPL.



Now you're ready to code! Let's keep going...

Creating and Editing Code

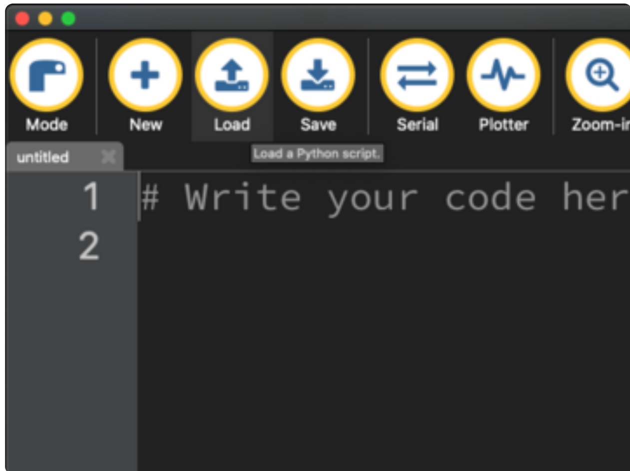
One of the best things about CircuitPython is how simple it is to get code up and running. This section covers how to create and edit your first CircuitPython program.

To create and edit code, all you'll need is an editor. There are many options. **Adafruit strongly recommends using Mu! It's designed for CircuitPython, and it's really simple and easy to use, with a built in serial console!**

If you don't or can't use Mu, there are a number of other editors that work quite well. The Recommended Editors page (https://adafru.it/Vue) has more details. Otherwise, make sure you do "Eject" or "Safe Remove" on Windows or "sync" on Linux after writing a file if you aren't using Mu. (This was formerly not a problem on macOS, but see the warning below.)

macOS Sonoma 14.1 introduced a bug that delays writes to small drives such as CIRCUITPY drives. This caused errors when saving files to CIRCUITPY. There is a workaround. The bug was fixed in Sonoma 14.4, but at the cost of greatly slowed writes to drives 1GB or smaller.

Creating Code



Installing CircuitPython generates a **code.py** file on your **CIRCUITPY** drive. To begin your own program, open your editor, and load the **code.py** file from the **CIRCUITPY** drive.

If you are using Mu, click the **Load** button in the button bar, navigate to the **CIRCUITPY** drive, and choose **code.py**.

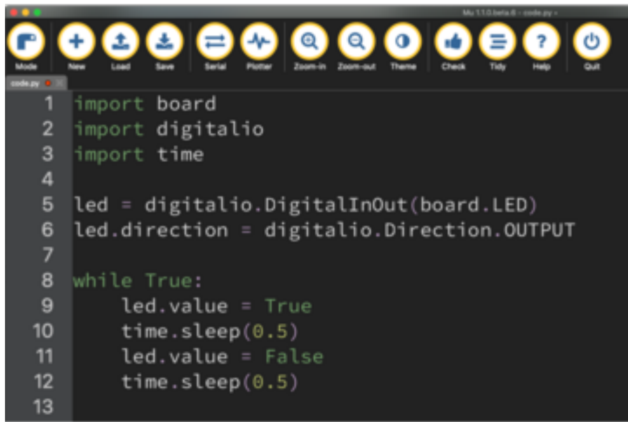
Copy and paste the following code into your editor:

```
import board
import digitalio
import time

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

while True:
    led.value = True
    time.sleep(0.5)
    led.value = False
    time.sleep(0.5)
```

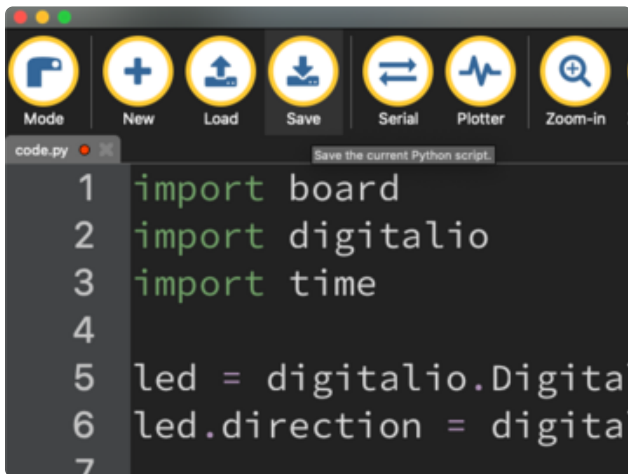
The KB2040, QT Py , Qualia, and the Trinkeys do not have a built-in little red LED! There is an addressable RGB NeoPixel LED. The above example will NOT work on the KB2040, QT Py, Qualia, or the Trinkeys!

If you're using a KB2040, QT Py, Quaila, or a Trinkey, or any other board without a single-color LED that can blink, please download the NeoPixel blink example (https://adafru.it/UDU).

The NeoPixel blink example uses the onboard NeoPixel, but the time code is the same. You can use the linked NeoPixel Blink example to follow along with this guide page.



It will look like this. Note that under the `while True:` line, the next four lines begin with four spaces to indent them, and they're indented exactly the same amount. All the lines before that have no spaces before the text.



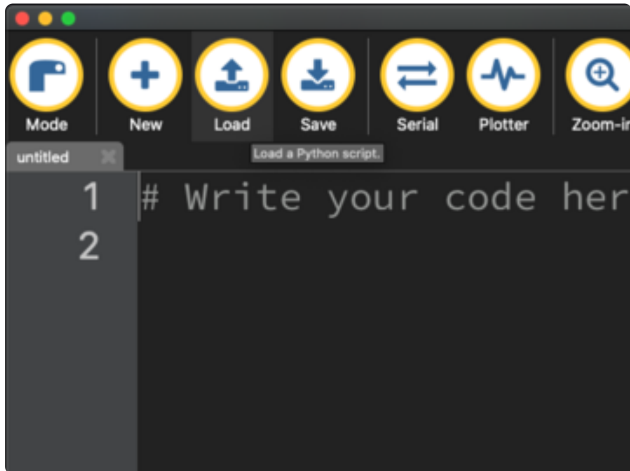
Save the **code.py** file on your **CIRCUITPY** drive.

The little LED should now be blinking. Once per half-second.

Congratulations, you've just run your first CircuitPython program!

On most boards you'll find a tiny red LED. On the ItsyBitsy nRF52840, you'll find a tiny blue LED. On QT Py M0, QT Py RP2040, Qualia, and the Trinkey series, you will find only an RGB NeoPixel LED.

Editing Code



To edit code, open the **code.py** file on your **CIRCUITPY** drive into your editor.

Make the desired changes to your code. Save the file. That's it!

Your code changes are run as soon as the file is done saving.

There's one warning before you continue...

Don't click reset or unplug your board!

The CircuitPython code on your board detects when the files are changed or written and will automatically re-start your code. This makes coding very fast because you save, and it re-runs. If you unplug or reset the board before your computer finishes writing the file to your board, you can corrupt the drive. If this happens, you may lose the code you've written, so it's important to backup your code to your computer regularly.

There are a couple of ways to avoid filesystem corruption.

1. Use an editor that writes out the file completely when you save it.

Check out the Recommended Editors page (https://adafru.it/Vue) for details on different editing options.

If you are dragging a file from your host computer onto the CIRCUITPY drive, you still need to do step 2. Eject or Sync (below) to make sure the file is completely written.

2. Eject or Sync the Drive After Writing

If you are using one of our not-recommended-editors, not all is lost! You can still make it work.

On Windows, you can Eject or Safe Remove the **CIRCUITPY** drive. It won't actually eject, but it will force the operating system to save your file to disk. On Linux, use the **sync** command in a terminal to force the write to disk.

You also need to do this if you use Windows Explorer or a Linux graphical file manager to drag a file onto **CIRCUITPY**.

Oh No I Did Something Wrong and Now The CIRCUITPY Drive Doesn't Show Up!!!

> Don't worry! Corrupting the drive isn't the end of the world (or your board!). If this happens, follow the steps found on the Troubleshooting (https://adafru.it/Den) page of every board guide to get your board up and running again.

If you are having trouble saving code on Windows 10, try including this code snippet at the top of code.py:

```
import supervisor
supervisor.runtime.autoreload = False
```

Back to Editing Code...

Now! Let's try editing the program you added to your board. Open your **code.py** file into your editor. You'll make a simple change. Change the first `0.5` to `0.1`. The code should look like this:

```
import board
import digitalio
import time

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

while True:
    led.value = True
    time.sleep(0.1)
```

```
    led.value = False
    time.sleep(0.5)
```

Leave the rest of the code as-is. Save your file. See what happens to the LED on your board? Something changed! Do you know why?

You don't have to stop there! Let's keep going. Change the second `0.5` to `0.1` so it looks like this:

```
while True:
    led.value = True
    time.sleep(0.1)
    led.value = False
    time.sleep(0.1)
```

Now it blinks really fast! You decreased the both time that the code leaves the LED on and off!

Now try increasing both of the `0.1` to `1`. Your LED will blink much more slowly because you've increased the amount of time that the LED is turned on and off.

Well done! You're doing great! You're ready to start into new examples and edit them to see what happens! These were simple changes, but major changes are done using the same process. Make your desired change, save it, and get the results. That's really all there is to it!

Naming Your Program File

CircuitPython looks for a code file on the board to run. There are four options: **code.txt**, **code.py**, **main.txt** and **main.py**. CircuitPython looks for those files, in that order, and then runs the first one it finds. While **code.py** is the recommended name for your code file, it is important to know that the other options exist. If your program doesn't seem to be updating as you work, make sure you haven't created another code file that's being read instead of the one you're working on.

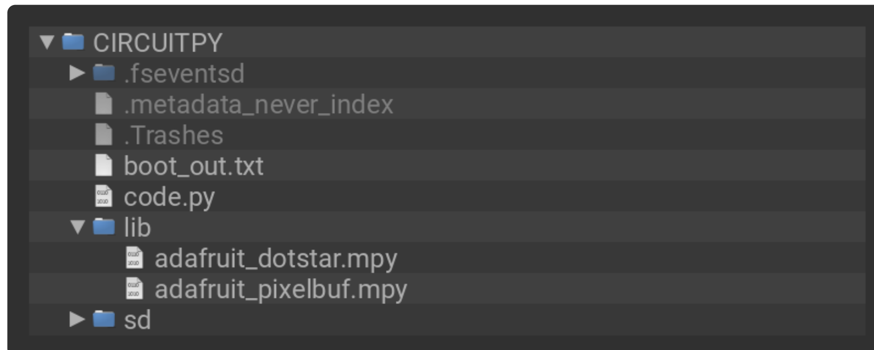
Code PyRuler with CircuitPython

Installing Project Code

To use with CircuitPython, you need to first install a few libraries, into the lib folder on your **CIRCUITPY** drive. Then you need to update **code.py** with the example script.

Thankfully, we can do this in one go. In the example below, click the **Download Project Bundle** button below to download the necessary libraries and the **code.py** file in a zip file. Extract the contents of the zip file, open the directory **PyRuler_Simon_Game/** and then click on the directory that matches the version of CircuitPython you're using and copy the contents of that directory to your **CIRCUITPY** drive.

Your **CIRCUITPY** drive should now look similar to the following image:



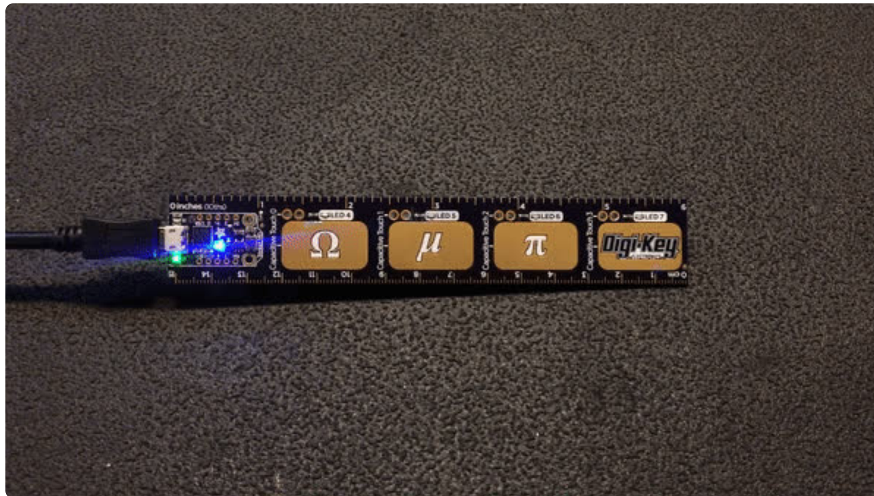

```
# SPDX-FileCopyrightText: 2019 Isaac Wellish for Adafruit Industries
#
# SPDX-License-Identifier: MIT

"""
This example runs the 'Simon' game on the PyRuler.
Memorize each led sequence and tap the corresponding
touch pads on the pyruler to advance to each new sequence.
Code adapted from Miguel Grinberg's Simon game for Circuit Playground Express

"""

import time
import random
import board
from rainbowio import colorwheel
from digitalio import DigitalInOut, Direction
import touchio
import adafruit_dotstar

# Initialize dot star led
pixels = adafruit_dotstar.DotStar(board.APA102_SCK, board.APA102_MOSI,
                                  1, brightness=0.1)
red = (255,0,0)
green = (0,255,0)
blue = (0,0,255)

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

touches = [DigitalInOut(board.CAP0)]
for p in (board.CAP1, board.CAP2, board.CAP3):
    touches.append(touchio.TouchIn(p))

leds = []
for p in (board.LED4, board.LED5, board.LED6, board.LED7):
    led = DigitalInOut(p)
    led.direction = Direction.OUTPUT
    leds.append(led)
```

```
cap_touches = [False, False, False, False]


def rainbow_cycle(wait):
    for j in range(255):
        for i in range(len(pixels)):
            rc_index = (i * 256 // len(pixels)) + j
            pixels[i] = colorwheel(rc_index & 255)
        time.sleep(wait)

def read_caps():
    t0_count = 0
    t0 = touches[0]
    t0.direction = Direction.OUTPUT
    t0.value = True
    t0.direction = Direction.INPUT
    # funky idea but we can 'diy' the one non-hardware captouch device by hand
    # by reading the drooping voltage on a tri-state pin.
    t0_count = t0.value + t0.value + t0.value + t0.value + t0.value + \
               t0.value + t0.value + t0.value + t0.value + t0.value + \
               t0.value + t0.value + t0.value + t0.value + t0.value
    cap_touches[0] = t0_count > 2
    cap_touches[1] = touches[1].raw_value > 3000
    cap_touches[2] = touches[2].raw_value > 3000
    cap_touches[3] = touches[3].raw_value > 3000
    return cap_touches

def timeout_touch(timeout=3):
    start_time = time.monotonic() # start 3 second timer waiting for user input
    while time.monotonic() - start_time < timeout:
        caps = read_caps()
        for i,c in enumerate(caps):
            if c:
                return i

def light_cap(cap, duration=0.5):
    # turn the LED for the selected cap on
    leds[cap].value = True
    time.sleep(duration)
    leds[cap].value = False
    time.sleep(duration)

def play_sequence(seq):
    duration = max(0.1, 1 - len(sequence) * 0.05)
    for cap in seq:
        light_cap(cap, duration)

def read_sequence(seq):
    pixels.fill(green)
    for cap in seq:
        if timeout_touch() != cap:
            # the player made a mistake!
            return False
        light_cap(cap, 0.5)
    return True

while True:
    # led light sequence at beginning of each game
    pixels.fill(blue)
    time.sleep(1)
    for led in leds:
        led.value = True
        time.sleep(0.25)
    for led in leds:
        led.value = False
    sequence = []
    while True:
        pixels.fill(blue) # blue for showing user sequence
```

```
        time.sleep(1)
        sequence.append(random.randint(0, 3)) # add new light to sequence each time
        play_sequence(sequence) # show the sequence
        if not read_sequence(sequence): # if user inputs wrong sequence, gameover
            # game over, make dot star red
            pixels.fill(red)
            time.sleep(3)
            print("gameover")
            break
        else:
            print("Next sequence unlocked!")
            rainbow_cycle(0) # Dot star animation after each correct sequence
        pixels.fill(0)
        time.sleep(1)
```

How Simon Works

Game play



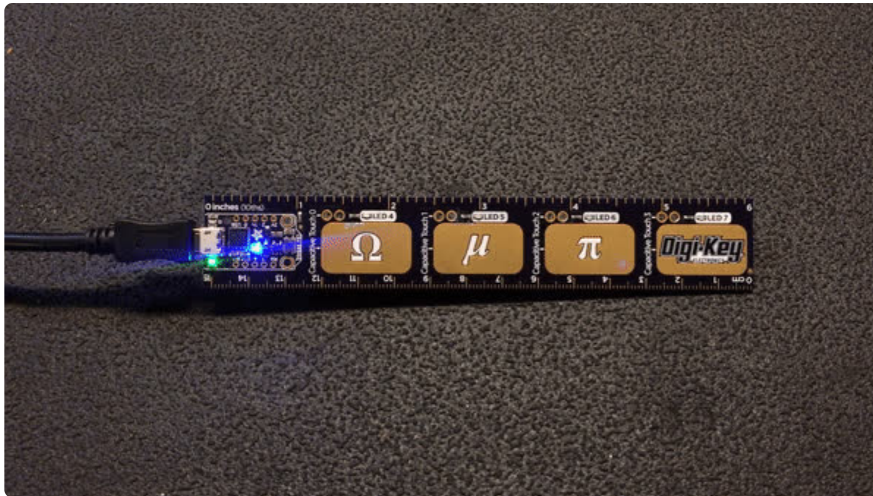
To play the Simon game, power up the PyRuler by plugging in a USB cable connected to a computer or a USB cell phone battery.

You will see the 4 leds above each touch pad light up in sequence then turn off.

Next, you will see one of the 4 leds turn on and off.

Then the DotStar RGB LED on the Trinket-sized circuit board will turn green prompting you to touch the pad corresponding to the led that was just on.

If you correctly input the given pattern, the DotStar led will cycle through a random pattern and the next pattern will be shown.



Each future pattern has the same initial sequence from before with one more added to the sequence each time.

When the DotStar is blue, the program is showing you the sequence.

If you get a sequence wrong, the DotStar will turn red then the game will start over with the leds cycling through the intro pattern.

How the code works

Here's what's going on behind the scenes to deliver you the Simon game you know and love.

Initialization

This segment towards the beginning of the program initializes the 4 touch pads on the PyRuler as well as the corresponding 4 leds.

```
touches = [DigitalInOut(board.CAP0)]
for p in (board.CAP1, board.CAP2, board.CAP3):
    touches.append(touchio.TouchIn(p))

leds = []
for p in (board.LED4, board.LED5, board.LED6, board.LED7):
    led = DigitalInOut(p)
    led.direction = Direction.OUTPUT
    leds.append(led)

cap_touches = [False, False, False, False]
```

Functions

These functions simplify and optimize how the program runs.

DotStar code

This part shows how the rainbow pattern for the dot star works.

```
def wheel(pos):
    # Input a value 0 to 255 to get a color value.
    # The colours are a transition r - g - b - back to r.
    if pos &lt; 0 or pos &gt; 255:
        return (0, 0, 0)
    if pos &lt; 85:
        return (255 - pos * 3, pos * 3, 0)
    if pos &lt; 170:
        pos -= 85
        return (0, 255 - pos * 3, pos * 3)
    pos -= 170
    return (pos * 3, 0, 255 - pos * 3)

def rainbow_cycle(wait):
    for j in range(255):
        for i in range(len(pixels)):
            rc_index = (i * 256 // len(pixels)) + j
            pixels[i] = wheel(rc_index &amp; 255)
        time.sleep(wait)
```

Detecting capacitive touch and setting a timer

Up next we read input from the PyRuler capacitive touch pads with the `read_caps()` function. Then with the `timeout_touch()` function we set a 3 second timer after each pattern is displayed as well as in-between each touch.

```
def read_caps():
    t0_count = 0
    t0 = touches[0]
    t0.direction = Direction.OUTPUT
    t0.value = True
    t0.direction = Direction.INPUT
    # funky idea but we can 'diy' the one non-hardware captouch device by hand
    # by reading the drooping voltage on a tri-state pin.
    t0_count = t0.value + t0.value + t0.value + t0.value + t0.value + \
               t0.value + t0.value + t0.value + t0.value + t0.value + \
               t0.value + t0.value + t0.value + t0.value + t0.value
    cap_touches[0] = t0_count &gt; 2
    cap_touches[1] = touches[1].raw_value &gt; 3000
    cap_touches[2] = touches[2].raw_value &gt; 3000
    cap_touches[3] = touches[3].raw_value &gt; 3000
    return cap_touches

def timeout_touch(timeout=3):
    start_time = time.monotonic() # start 3 second timer waiting for user input
    while time.monotonic() - start_time &lt; timeout:
        caps = read_caps()
```

```
        for i,c in enumerate(caps):
            if c:
                return i
```

Playing and reading each sequence with leds

- `light_cap()` turns on the led associated with each touch pad if the pad was touched.
- `play_sequence()` plays each led for the given sequence and slowly speeds up the playback of each sequence as they get longer.
- `read_sequence()` First turns the DotStar green (indicating to user to enter the sequence) then reads the touch pads and determines if they are the right sequence. If the wrong pad was touched, the function returns `False` which will cause a game over (more on this later).

```
def light_cap(cap, duration=0.5):
    # turn the LED for the selected cap on
    leds[cap].value = True
    time.sleep(duration)
    leds[cap].value = False
    time.sleep(duration)

def play_sequence(seq):
    duration = max(0.1, 1 - len(sequence) * 0.05)
    for cap in seq:
        light_cap(cap, duration)

def read_sequence(seq):
    pixels.fill(green)
    for cap in seq:
        if timeout_touch() != cap:
            # the player made a mistake!
            return False
        light_cap(cap, 0.5)
    return True
```

The main loop

- First trigger the starting sequence of leds demonstrating the game is beginning.
- Next in a nested loop, turn the DotStar blue demonstrating the sequence is being shown.
- Then add a random number between 0 and 3 to the sequence and play the sequence.
    - If the user enters the wrong sequence or the time runs out, turn the DotStar red indicating game over, and exit the loop starting the game over at the top of the main loop.
    - Otherwise, trigger the rainbow animation on the DotStar (indicating a correct sequence) and move to next sequence.

```
while True:
    # led light sequence at beginning of each game
    pixels.fill(blue)
    time.sleep(1)
    for led in leds:
        led.value = True
        time.sleep(0.25)
    for led in leds:
        led.value = False
    sequence = []
    while True:
        pixels.fill(blue) # blue for showing user sequence
        time.sleep(1)
        sequence.append(random.randint(0, 3)) # add new light to sequence each time
        play_sequence(sequence) # show the sequence
        if not read_sequence(sequence): # if user inputs wrong sequence, gameover
            # game over, make dot star red
            pixels.fill(red)
            time.sleep(3)
            print("gameover")
            break
        else:
            print("Next sequence unlocked!")
            rainbow_cycle(0) # Dot star animation after each correct sequence
        pixels.fill(0)
        time.sleep(1)
```

That's it, now you're a CircuitPython wiz!

Simon Says time to make your own game with the PyRuler!