Pyloton: CircuitPython Cycling Computer

Created by John Park

https://learn.adafruit.com/pyloton

Last updated on 2023-08-29 04:22:14 PM EDT
# Table of Contents

**Overview**  
- Parts List  
- Parts  
- Heart Rate Monitor  
- iOS Device  

**Understanding BLE**  
- BLE Basics  
- Bluetooth LE Terms  

**CircuitPython on CLUE**  
- Set up CircuitPython Quick Start!  

**Code the Pyloton in CircuitPython for CLUE**  
- Code.py  
- Code Run Through  
- Pyloton.py Run Through  

**3D Printing**  
- 3D Printed Parts  
- Slicing Parts  

**Use the Pyloton**  
- iOS Connection  
- AMS Connection  
- Sensor Connections  
- Display  
- Get Track Info  
- Send Media Control Commands  

©Adafruit Industries
Overview

Pyloton is a CircuitPython bike computer made with the CLUE board. Pyloton measures Bluetooth LE heart rate, speed, and cadence.

It also provides Apple Music Service song info, combined on one small device with a sharp display and a 3D printed handlebar mount (or optional wrist mount).

This project is the culmination of three previous Adafruit Learning System projects, plus some new abilities of our Bluefruit code to connect to multiple peripheral devices!
Previously, we created these standalone projects:

- **Now Playing: Bluetooth LE Apple Media Service Display**
- **BLE Heart Rate Zone Trainer Display**
- **BLE Cycling Speed & Cadence Sensor Display**

You may refer to those guides for additional details on the sensors, libraries, and code. Here, we'll show you how they can all be combined into one device.
Parts List

CLUE ()
420mAh battery () or
400mAh ()
350mAh ()
Slide Switch ()
JST Extension ()
Two 3/8 to 1/4-20 thread ()
1/4” to 1/4” Screw Adapter ()
Swivel-Head Pan Tilt ()
M5x16mm screw + nut
Wahoo sensor ()
PLA or ABS Filament ()
Parts

Adafruit CLUE - nRF52840 Express with Bluetooth LE
Do you feel like you just don't have a CLUE? Well, we can help with that - get a CLUE here at Adafruit by picking up this sensor-packed development board. We wanted to build some...
https://www.adafruit.com/product/4500

Lithium Ion Polymer Battery with Short Cable - 3.7V 420mAh
Lithium-ion polymer (also known as 'lipo' or 'lipoly') batteries are thin, light, and powerful. The output ranges from 4.2V when completely charged to 3.7V. This...
https://www.adafruit.com/product/4236

Breadboard-friendly SPDT Slide Switch
These nice switches are perfect for use with breadboard and perfboard projects. They have 0.1" spacing and snap in nicely into a solderless breadboard. They're easy to switch...
https://www.adafruit.com/product/805
JST-PH Battery Extension Cable - 500mm
By popular demand, we now have a handy extension cord for all of our JST PH-terminated battery packs (such as our LiIon/LiPoly and 3xAAA holders). One end has a JST-PH compatible...
https://www.adafruit.com/product/1131

Third-Party Products

Heart Rate Monitor
You’ll need a heart rate monitor that supports Bluetooth Low Energy (BLE). I'm using the Scosche RHYTHM+ () but you should be able to use any monitor that uses the Bluetooth SIG Heart Rate service standard.

These work by flashing green (and sometimes yellow) LEDs against your skin and then measuring the reflected light that returns. The color changes/darkens during the pulse of your heart thanks to all that blood sloshing around!
Cycling Speed & Cadence Sensor
Bluetooth LE compatible, such as the Wahoo Fitness Blue SC. This type of device is two sensors in one package and typically reads speed and cadence revolutions based on a pair of magnets affixed to a spoke and crank.

You can also use individual speed and/or cadence sensors that use an IMU rather than a magnet to sense revolutions, such as the Wahoo RPM Speed and RPM Cadence.

Be sure that your sensors use Bluetooth LE and not Ant+ or some other radio standard. (Some use both, which is fine.)
iOS Device
You'll also need an iPhone or iPod touch with BLE capabilities if you want to see song playback info on the Pyloton.

Adafruit intends to have a comparable Android application for Bluetooth monitoring. There currently is no time frame for the release of an app. We regret the inconvenience.

Understanding BLE

BLE Basics

To understand how we communicate between the MagicLight Bulb and the Circuit Playground Bluefruit (CPB), it's first important to get an overview of how Bluetooth Low Energy (BLE) works in general.

The nRF52840 chip on the CPB uses Bluetooth Low Energy, or BLE. BLE is a wireless communication protocol used by many devices, including mobile devices. You can communicate between your CPB and peripherals such as the Magic Light, mobile devices, and even other CPB boards!

There are a few terms and concepts commonly used in BLE with which you may want to familiarize yourself. This will help you understand what your code is doing when you're using CircuitPython and BLE.
Two major concepts to know about are the two modes of BLE devices:

- Broadcasting mode (also called GAP for Generic Access Profile)
- Connected device mode (also called GATT for Generic Attribute Profile).

GAP mode deals with broadcasting peripheral advertisements, such as "I'm a device named LEDBlue-19592CBC", as well as advertising information necessary to establish a dedicated device connection if desired. The peripheral may also be advertising available services.

GATT mode deals with communications and attribute transfer between two devices once they are connected, such as between a heart monitor and a phone, or between your CPB and the Magic Light.

Bluetooth LE Terms

GAP Mode

Device Roles:

- Peripheral - The low-power device that broadcasts advertisements. Examples of peripherals include: heart rate monitor, smart watch, fitness tracker, iBeacon, and the Magic Light. The CPB can also work as a peripheral.
• Central - The host "computer" that observes advertisements being broadcast by
the Peripherals. This is often a mobile device such as a phone, tablet, desktop
or laptop, but the CPB can also act as a central (which it will in this project).

Terms:

• Advertising - Information sent by the peripheral before a dedicated connection
has been established. All nearby Centrals can observe these advertisements. When a peripheral device advertises, it may be transmitting the name of the
device, describing its capabilities, and/or some other piece of data. Central can
look for advertising peripherals to connect to, and use that information to
determine each peripheral's capabilities (or Services offered, more on that
below).

GATT Mode

Device Roles:

• Server - In connected mode, a device may take on a new role as a Server,
providing a Service available to clients. It can now send and receive data
packets as requested by the Client device to which it now has a connection.
• Client - In connected mode, a device may also take on a new role as Client that
can send requests to one or more of a Server's available Services to send and
receive data packets.

NOTE: A device in GATT mode can take on the role of both Server and Client
while connected to another device.

Terms:

• Profile - A pre-defined collection of Services that a BLE device can provide. For example, the Heart Rate Profile, or the Cycling Sensor (bike computer) Profile. These Profiles are defined by the Bluetooth Special Interest Group (SIG). For devices that don't fit into one of the pre-defined Profiles, the manufacturer
creates their own Profile. For example, there is not a "Smart Bulb" profile, so the
Magic Light manufacturer has created their own unique one.
• Service - A function the Server provides. For example, a heart rate monitor
armband may have separate Services for Device Information, Battery Service,
and Heart Rate itself. Each Service is comprised of collections of information
called Characteristics. In the case of the Heart Rate Service, the two
Characteristics are Heart Rate Measurement and Body Sensor Location. The peripheral advertises its services.

- **Characteristic** - A Characteristic is a container for the value, or attribute, of a piece of data along with any associated metadata, such as a human-readable name. A characteristic may be readable, writable, or both. For example, the Heart Rate Measurement Characteristic can be served up to the Client device and will report the heart rate measurement as a number, as well as the unit string "bpm" for beats-per-minute. The Magic Light Server has a Characteristic for the RGB value of the bulb which can be written to by the Central to change the color. Characteristics each have a Universal Unique Identifier (UUID) which is a 16-bit or 128-bit ID.

- **Packet** - Data transmitted by a device. BLE devices and host computers transmit and receive data in small bursts called packets.

This guide is another good introduction to the concepts of BLE, including GAP, GATT, Profiles, Services, and Characteristics.

---

**CircuitPython on CLUE**

CircuitPython is a derivative of MicroPython 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. Simply copy and edit files on the CIRCUITPY flash drive to iterate.

The following instructions will show you how to install CircuitPython. If you've already installed CircuitPython but are looking to update it or reinstall it, the same steps work for that as well!

**Set up CircuitPython Quick Start!**

Follow this quick step-by-step for super-fast Python power :)
Click the link above to download the latest version of CircuitPython for the CLUE.

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

Plug your CLUE 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 Reset button on the top (magenta arrow) on your board, and you will see the NeoPixel RGB LED (green arrow) turn green. If it turns red, check the USB cable, try another USB port, etc. Note: The little red LED next to the USB connector will pulse red. That's 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 CLUEBOOT.

Drag the adafruit-circuitpython-clue-etc.uf2 file to CLUEBOOT.

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

If this is the first time you're installing CircuitPython or you're doing a completely fresh install after erasing the filesystem, you will have two files - boot_out.txt, and code.py, and one folder - lib on your CIRCUITPY drive.

If CircuitPython was already installed, the files present before reloading CircuitPython should still be present on your CIRCUITPY drive. Loading CircuitPython will not create new files if there was already a CircuitPython filesystem present.

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

©Adafruit Industries
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 CircuitPython_Pyloton on/ 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: 2020 Eva Herrada for Adafruit Industries
#
# SPDX-License-Identifier: MIT

from time import time
import adafruit_ble
import board
import pyloton

ble = adafruit_ble.BLERadio()  # pylint: disable=no-member

CONNECTION_TIMEOUT = 45

HEART = True
```
```python
SPEED = True
CADENCE = True
AMS = True
DEBUG = False

# 84.229 is wheel circumference (700x23 in my case)
pyloton = pyloton.Pyloton(ble, board.DISPLAY, 84.229, HEART, SPEED, CADENCE, AMS, DEBUG)

pyloton.show_splash()

ams = pyloton.ams_connect()

start = time()
hr_connection = None
speed_cadence_connections = []
while True:
    if HEART:
        if not hr_connection:
            print("Attempting to connect to a heart rate monitor")
            hr_connection = pyloton.heart_connect()
            ble.stop_scan()
        if SPEED or CADENCE:
            if not speed_cadence_connections:
                print("Attempting to connect to speed and cadence monitors")
                speed_cadence_connections = pyloton.speed_cadence_connect()

    if time()-start >= CONNECTION_TIMEOUT:
        pyloton.timeout()
        break
    # Stop scanning whether or not we are connected.
    ble.stop_scan()

    #pylint: disable=too-many-boolean-expressions
    if ((not HEART or (hr_connection and hr_connection.connected)) and
        ((not SPEED and not CADENCE) or
         (speed_cadence_connections and speed_cadence_connections[0].connected))
    and
        (not AMS or (ams and ams.connected))):
        break

    pyloton.setup_display()

    while ((not HEART or hr_connection.connected) and
            ((not SPEED or not CADENCE) or
             (speed_cadence_connections and speed_cadence_connections[0].connected)) and
            (not AMS or (ams and ams.connected))):
        pyloton.update_display()
        pyloton.ams_remote()

    print("\n\nNot all sensors are connected. Please reset to try again\n\n")

Then, you can click on the Download: Project Zip link in the window below to
download the code file and .STL file.

**Code Run Through**

First, the code loads all the required libraries.

```
import board
import pyloton

Pyloton Setup

Then, the variables required to run Pyloton are defined. To disable a specific sensor, set that sensor's variable to False. For example, if I wanted to use AppleMediaServices and a speed and cadence sensor, but didn't want to use a heart rate monitor, I would set HEART to False and this section of code.py would look like this:

```python
HEART = False
SPEED = True
CADENCE = True
AMS = True
DEBUG = False
```

After that, Pyloton is instantiated and all the required variables are passed into it. My bike has 700x23 tires on it, so my wheel diameter is 84.229 inches. To calculate your wheel circumference, use the chart [here](#) and then convert the diameter to inches. There are 2.54 centimeters in one inch if you need to convert centimeters to inches.

```python
ble = adafruit_ble.BLERadio()    # pylint: disable=no-member
CONNECTION_TIMEOUT = 45
HEART = True
SPEED = True
CADENCE = True
AMS = True
DEBUG = False

# 84.229 is wheel circumference (700x23 in my case)
pyloton = pyloton.Pyloton(ble, board.DISPLAY, 84.229, HEART, SPEED, CADENCE, AMS, DEBUG)
```

Preparing to Connect

At this stage, we first tell Pyloton to show the splash screen so we can see the status of Pyloton easily.
Then, we tell Pyloton to connect to an Apple device. Simply comment out that line and disable AMS if you don’t intend on using one.

After that, we set the variable `start` to the current time so that the code can tell when it started and time out if it’s taking too long.

Then, `hr_connection` and `speed_cadence_connections` are set to `False` so that they don’t cause a `NameError` when the setup loop checks the status of the various sensors.

```python
pyloton.show_splash()
ams = pyloton.ams_connect()
start = time()
hr_connection = None
speed_cadence_connections = []
```

## Connecting loop

This **While** loop first checks if specific sensors are already connected. If they aren’t, and that sensor is enabled, it will run the functions in `pyloton.py` to try to connect to those sensors.

Then, if more time than the desired timeout length has passed, 45 seconds by default, it will break out of the loop and code.py will finish running.

After that, a check if a sensor is connected only if it is enabled. If all enabled sensors are connected, break out of the loop and go to the next section.

```python
while True:
    if HEART:
        if not hr_connection:
            print("Attempting to connect to a heart rate monitor")
            hr_connection = pyloton.heart_connect()
            ble.stop_scan()
    if SPEED or CADENCE:
        if not speed_cadence_connections:
            print("Attempting to connect to speed and cadence monitors")
            speed_cadence_connections = pyloton.speed_cadence_connect()
    if time()-start &gt;= CONNECTION_TIMEOUT:
        pyloton.timeout()
        break
    # Stop scanning whether or not we are connected.
    ble.stop_scan()

    #pylint: disable=too-many-boolean-expressions
    if ((not HEART or (hr_connection and hr_connection.connected)) and
        ((not SPEED and not CADENCE) or
         (speed_cadence_connections and speed_cadence_connections[0].connected))
        and
```
Main loop

This loop uses the same logic as the lines above to determine if all enabled sensors are connected. If they are, it then updates the display and runs the `ams.remote` function. Comment out that line if you aren't using a device with AppleMediaServices.

If one of the sensors disconnects, it breaks the loop and ends the file, notifying the user that not all sensors are connected.

```python
while ((not HEART or hr_connection.connected) and
        ((not SPEED or not CADENCE) or
         (speed_cadence_connections and speed_cadence_connections[0].connected)) and
        (not AMS or ams.connected)):
    pyloton.update_display()
    pyloton.ams_remote()
    print("\n\nNot all sensors are connected. Please reset to try again\n\n")
```

Pyloton.py Run Through

To start off, we import all of Pyloton's dependencies.

```python
import time
import adafruit_bitmap_font
from adafruit_bitmap_font import bitmap_font
from adafruit_display_shapes import Rect
from adafruit_display_text import label
from adafruit_ble import AppleMediaService
from adafruit_ble_apple_media import UnsupportedCommand
import gamepad
import touchio
```

Clue library

Pyloton contains a small section of the [Adafruit_CircuitPython_CLUE library](https://circuitpython.readthedocs.io/en/latest/adafruit_circuitpython_CLUE/). We do this because the CLUE library has a lot of great tools for getting data from sensors, but for Pyloton we only need functions to get data from the buttons and capacitive touch pads. The CLUE library is documented [here](https://circuitpython.readthedocs.io/en/latest/adafruit_circuitpython_CLUE/).
class Clue:
    ""
    A very minimal version of the CLUE library.
    The library requires the use of many sensor-specific
    libraries this project doesn't use, and they were
taking up a lot of RAM.
    ""
    def __init__(self):
        self._i2c = board.I2C()
        self._touches = [board.D0, board.D1, board.D2]
        self._touch_threshold_adjustment = 0
        self._a = digitalio.DigitalInOut(board.BUTTON_A)
        self._a.switch_to_input(pull=digitalio.Pull.UP)
        self._b = digitalio.DigitalInOut(board.BUTTON_B)
        self._b.switch_to_input(pull=digitalio.Pull.UP)
        self._gamepad = gamepad.GamePad(self._a, self._b)

    @property
def were_pressed(self):
        ""
        Returns a set of buttons that have been pressed since the last time
        were_pressed was run.
        ""
        ret = set()
        pressed = self._gamepad.get_pressed()
        for button, mask in (('@', 0x01), ('B', 0x02)):
            if mask & pressed:
                ret.add(button)
        return ret

    def _touch(self, i):
        if not isinstance(self._touches[i], touchio.TouchIn):
            self._touches[i] = touchio.TouchIn(self._touches[i])
            self._touches[i].threshold += self._touch_threshold_adjustment
        return self._touches[i].value

    @property
def touch_0(self):
        ""
        Returns True when capacitive touchpad 0 is currently being pressed.
        ""
        return self._touch(0)

    @property
def touch_1(self):
        ""
        Returns True when capacitive touchpad 1 is currently being pressed.
        ""
        return self._touch(1)

    @property
def touch_2(self):
        ""
        Returns True when capacitive touchpad 2 is currently being pressed.
        ""
        return self._touch(2)

Pyloton Class

Here, Pyloton is defined. We then create some class variables used for setting the
colors in the UI and create a Clue object.
class Pyloton:
    
    Contains the various functions necessary for doing the Pyloton learn guide.
    
    #pylint: disable=too-many-instance-attributes

    YELLOW = 0xFCFF00
    PURPLE = 0x64337E
    WHITE = 0xFFFFFF

    clue = Clue()

__init__

**Init** first defines all of the required instance variables. It then loads fonts, loads the sprite sheet, and sets up the group used to display status update text.

def __init__(self, ble, display, circ, heart=True, speed=True, cad=True, ams=True, debug=False): #pylint: disable=too-many-arguments
    self.debug = debug
    self.ble = ble
    self.display = display
    self.circumference = circ

    self.heart_enabled = heart
    self.speed_enabled = speed
    self.cadence_enabled = cad
    self.ams_enabled = ams
    self.hr_connection = None
    self.num_enabled = heart + speed + cad + ams

    self._previous_wheel = 0
    self._previous_crank = 0
    self._previous_revolutions = 0
    self._previous_rev = 0
    self._previous_speed = 0
    self._previous_cadence = 0
    self._previous_heart = 0
    self._speed_failed = 0
    self._cadence_failed = 0
    self._setup = 0
    self._hr_label = None
    self._sp_label = None
    self._cadence_label = None
    self._ams_label = None
    self._hr_service = None
    self._heart_y = None
    self._speed_y = None
    self._cadence_y = None
    self._ams_y = None
    self.ams = None
    self.cyc_connections = None
    self.cyc_services = None
    self.track_artist = True
    self.start = time.time()

    self.splash = displayio.Group()
    self.loading_group = displayio.Group()

    self._load_fonts()
self.sprite_sheet, self.palette = adafruit_imageload.load("/sprite_sheet.bmp",
bitmap=displayio.Bitmap,
palette=displayio.Palette)
self.text_group = displayio.Group()
self.status = label.Label(font=self.arial12, x=10, y=200, 
text='', color=self.YELLOW)
self.status1 = label.Label(font=self.arial12, x=10, y=220, 
text='', color=self.YELLOW)
self.text_group.append(self.status)
self.text_group.append(self.status1)

show_splash

Show splash is used to load and display the splash screen (essentially a loading
screen) using displayio. If debug is enabled, the splash screen will not be displayed.

def show_splash(self):
    """
    Shows the loading screen
    """
    if self.debug:
        return

    blinka_bitmap = "blinka-pyloton.bmp"

    # Compatible with CircuitPython 6 & 7
    with open(blinka_bitmap, 'rb') as bitmap_file:
        bitmap1 = displayio.OnDiskBitmap(bitmap_file)
        tile_grid = displayio.TileGrid(bitmap1, pixel_shader=getattr(bitmap1, 
        'pixel_shader', displayio.ColorConverter()))
        self.loading_group.append(tile_grid)
        self.display.show(self.loading_group)
        status_heading = label.Label(font=self.arial16, x=80, y=175, 
        text="Status", color=self.YELLOW)
        rect = Rect(0, 165, 240, 75, fill=self.PURPLE)
        self.loading_group.append(rect)
        self.loading_group.append(status_heading)

    # # Compatible with CircuitPython 7+
    # bitmap1 = displayio.OnDiskBitmap(blinka_bitmap)
    # tile_grid = displayio.TileGrid(bitmap1, pixel_shader=bitmap1.pixel_shader)
    # self.loading_group.append(tile_grid)
    # self.display.show(self.loading_group)
    # status_heading = label.Label(font=self.arial16, x=80, y=175, 
    # text="Status", color=self.YELLOW)
    # rect = Rect(0, 165, 240, 75, fill=self.PURPLE)
    # self.loading_group.append(rect)
    # self.loading_group.append(status_heading)

_load_fonts

_load_fonts does two things. First, it loads the 3 fonts we will be using, 12 pt. Arial, 
16 pt. Arial, and 24 pt. Arial bold. Then, it loads a bytestring of commonly used
characters. Loading these characters ahead of time is not required, however it does speed up the display process quite a bit.

```python
def _load_fonts(self):
    """
    Loads fonts
    """
    self.arial12 = bitmap_font.load_font("/fonts/Arial-12.bdf")
    self.arial16 = bitmap_font.load_font("/fonts/Arial-16.bdf")
    self.arial24 = bitmap_font.load_font("/fonts/Arial-Bold-24.bdf")
    glyphs = b'0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ-!,. "'?!'
    self.arial12.load_glyphs(glyphs)
    self.arial16.load_glyphs(glyphs)
    self.arial24.load_glyphs(glyphs)
```

__status_update__

__status_update__ allows editing the text of `self.status` and `self.status1` to display the current status update on the CLUE's screen. When message is longer than 25 characters, it gets sliced, with the first 25 characters appearing on the first line and the 25th through 50th characters appearing on the second line. If `debug` is enabled, it will simply print the full message to the REPL.

```python
def _status_update(self, message):
    """
    Displays status updates
    """
    if self.debug:
        print(message)
        return
    if self.text_group not in self.loading_group:
        self.loading_group.append(self.text_group)
    self.status.text = message[:25]
    self.status1.text = message[25:50]
```

timeout

This function is called by code.py when the `CONNECTION_TIMEOUT` has been reached. It displays a message that the program has timed out for three seconds, and then the program presumably ends.

```python
def timeout(self):
    """
    Displays Timeout on screen when pyloton has been searching for a sensor for too long
    """
    self._status_update("Pyloton: Timeout")
    time.sleep(3)
```
heart_connect

Heart connect uses the `circuitpython_ble_heart_rate()` library to connect to a heart rate monitor. This is a modified version of the `ble_heart_rate_simpletest.py` file.

```python
def heart_connect(self):
    """
    Connects to heart rate sensor
    """
    self._status_update("Heart Rate: Scanning...")
    for adv in self.ble.start_scan(ProvideServicesAdvertisement, timeout=5):
        if HeartRateService in adv.services:
            self._status_update("Heart Rate: Found an advertisement")
            self.hr_connection = self.ble.connect(adv)
            self._status_update("Heart Rate: Connected")
            break
    self.ble.stop_scan()
    if self.hr_connection:
        self._hr_service = self.hr_connection[HeartRateService]
        return self.hr_connection
```

ams_connect

This function is used to connect to an Apple device using the `ble_apple_media()` library. It is based on the example code from `ble_apple_media_simpletest.py`.

```python
def ams_connect(self, start=time.time(), timeout=30):
    """
    Connect to an Apple device using the ble_apple_media library
    """
    self._status_update("AppleMediaService: Connect your phone now")
    radio = adafruit_ble.BLERadio()
    a = SolicitServicesAdvertisement()
    a.solicited_services.append(AppleMediaService)
    radio.start_advertising(a)
    while not radio.connected and not self._has_timed_out(start, timeout):
        pass
    self._status_update("AppleMediaService: Connected")
    for connection in radio.connections:
        if not connection.paired:
            connection.pair()
        self._status_update("AppleMediaService: Paired")
        self.ams = connection[AppleMediaService]
    return radio
```

speed_cadence_connect

This function uses the `ble_cycling_speed_and_cadence()` library to connect to speed and cadence sensors. It works with both discrete speed and cadence sensors and combined speed and cadence sensors. It will not work if you were to connect two of the same sensor type, as in two speed sensors or a speed and cadence sensor and a
speed sensor. This is based on the example file, `ble_cycling_speed_and_cadence_simpletest.py`.

```python
def speed_cadence_connect(self):
    """
    Connects to speed and cadence sensor
    """
    self._status_update("Speed and Cadence: Scanning...")
    # Save advertisements, indexed by address
    advs = {}
    for adv in self.ble.start_scan(ProvideServicesAdvertisement, timeout=5):
        if CyclingSpeedAndCadenceService in adv.services:
            self._status_update("Speed and Cadence: Found an advertisement")
            # Save advertisement. Overwrite duplicates from same address
            advs[adv.address] = adv
    self.ble.stop_scan()
    self._status_update("Speed and Cadence: Stopped scanning")
    if not advs:
        # Nothing found. Go back and keep looking.
        return []
    # Connect to all available CSC sensors.
    self.cyc_connections = []
    for adv in advs.values():
        self.cyc_connections.append(self.ble.connect(adv))
        self._status_update("Speed and Cadence: Connected
{}".format(len(self.cyc_connections)))
    self.cyc_services = []
    for conn in self.cyc_connections:
        self.cyc_services.append(conn[CyclingSpeedAndCadenceService])
        self._status_update("Pyloton: Finishing up...")
    return self.cyc_connections
```

```python
def _compute_speed(self, values, speed):
    wheel_diff = values.last_wheel_event_time - self._previous_wheel
    rev_diff = values.cumulative_wheel_revolutions - self._previous_revolutions
    if wheel_diff:
        # Rotations per minute is 60 times the amount of revolutions since
        # the last update over the time since the last update
        rpm = 60*(rev_diff/(wheel_diff/1024))
        # We then multiply it by the wheel's circumference and convert it to mph
        speed = round((rpm * self.circumference) * (60/63360), 1)
        if speed < 0:
            speed = self._previous_speed
        self._previous_speed = speed
    else:
        self._speed_failed = 0
```

`_compute_speed`

A function used to turn the total number of revolutions and the time since the last revolution into speed in miles per hour. The speed is computed by converting revolutions since the last report, `rev_diff`, and time since the last report, `wheel_diff` into rotations per minute and then into miles per hour using some fancy unit conversion.
self._speed_failed += 1
if self._speed_failed &gt;= 3:
    speed = 0
self._previous_wheel = values.last_wheel_event_time

return speed

__compute_cadence

This works in a very similar way to __compute_speed, except since cadence is measured in rotations per minute, it doesn't need as much unit conversion. Also similar to the previous function, most of the function is related to preventing undesired results being displayed.

def __compute_cadence(self, values, cadence):
    crank_diff = values.last_crank_event_time - self._previous_crank
    crank_rev_diff = values.cumulative_crank_revolutions-self._previous_rev

    if crank_rev_diff:
        # Rotations per minute is 60 times the amount of revolutions since the
        # last update over the time since the last update
        cadence = round(60*(crank_rev_diff/(crank_diff/1024)), 1)
        if cadence &lt; 0:
            cadence = self._previous_cadence
            self._previous_cadence = cadence
            self._previous_rev = values.cumulative_crank_revolutions
            self._cadence_failed = 0
        else:
            self._cadence_failed += 1
            if self._cadence_failed &gt;= 3:
                cadence = 0
                self._previous_crank = values.last_crank_event_time
    return cadence

Read speed and cadence

Reads speed and cadence. Will set speed and/or cadence to 0 if they haven't changed in around the last 2 seconds. Also sets them to their previous values if it receives None in svc.measurement_values. Additionally, it will truncate speed and cadence to ensure they don't fill up their labels.

def read_s_and_c(self):
    
    speed = self._previous_speed
    cadence = self._previous_cadence
    for conn, svc in zip(self.cyc_connections, self.cyc_services):
        if not conn.connected:
            speed = cadence = 0
            continue
        values = svc.measurement_values
        if not values:
            if self._cadence_failed &gt;= 3 or self._speed_failed &gt;= 3:
                if self._cadence_failed &gt;= 3:
This works in a very similar way to __compute_speed, except since cadence is measured in rotations per minute, it doesn't need as much unit conversion. Also similar to the previous function, most of the function is related to preventing undesired results being displayed.

def __compute_cadence(self, values, cadence):
    crank_diff = values.last_crank_event_time - self._previous_crank
    crank_rev_diff = values.cumulative_crank_revolutions-self._previous_rev

    if crank_rev_diff:
        # Rotations per minute is 60 times the amount of revolutions since the
        # last update over the time since the last update
        cadence = round(60*(crank_rev_diff/(crank_diff/1024)), 1)
        if cadence &lt; 0:
            cadence = self._previous_cadence
            self._previous_cadence = cadence
            self._previous_rev = values.cumulative_crank_revolutions
            self._cadence_failed = 0
        else:
            self._cadence_failed += 1
            if self._cadence_failed &gt;= 3:
                cadence = 0
                self._previous_crank = values.last_crank_event_time
    return cadence

Read speed and cadence

Reads speed and cadence. Will set speed and/or cadence to 0 if they haven't changed in around the last 2 seconds. Also sets them to their previous values if it receives None in svc.measurement_values. Additionally, it will truncate speed and cadence to ensure they don't fill up their labels.

def read_s_and_c(self):
    
    speed = self._previous_speed
    cadence = self._previous_cadence
    for conn, svc in zip(self.cyc_connections, self.cyc_services):
        if not conn.connected:
            speed = cadence = 0
            continue
        values = svc.measurement_values
        if not values:
            if self._cadence_failed &gt;= 3 or self._speed_failed &gt;= 3:
                if self._cadence_failed &gt;= 3:
cadence = 0
if self._speed_failed &gt; 3:
speed = 0
    continue
if not values.last_wheel_event_time:
    continue
speed = self._compute_speed(values, speed)
if not values.last_crank_event_time:
    continue
cadence = self._compute_cadence(values, cadence)

if speed:
speed = str(speed)[:8]
if cadence:
cadence = str(cadence)[:8]

return speed, cadence

read_heart

Reads data from the heart rate monitor. Occasionally, it sends None as the current
heart rate, and in that case, we set the current heart rate to the previous heart rate
and return it.

```python
def read_heart(self):
    """
    Reads data from the heart rate sensor
    """
    measurement = self._hr_service.measurement_values
    if measurement is None:
        heart = self._previous_heart
    else:
        heart = measurement.heart_rate
        self._previous_heart = measurement.heart_rate

    if heart:
        heart = str(heart)[:4]
    return heart
```

read_ams

Gets data about the current playing track from AppleMediaServices. Every 3 seconds,
this switches from being the artist to being the track title. If either of these is longer
than 16 characters, it gets truncated to prevent the label from filling up.

```python
def read_ams(self):
    """
    Reads data from AppleMediaServices
    """
    current = time.time()
    try:
        if current - self.start &gt; 3:
            self.track_artist = not self.track_artist
            self.start = time.time()
        if self.track_artist:
            data = self.ams.artist
            if not self.track_artist:
                self.track_artist = not self.track_artist
                self.start = time.time()
data = self.ams.title
except (RuntimeError, UnicodeError):
data = None

if data:
data = data[:16] + (data[16:] and '..')
return data

_icon_maker

_icon_maker is used to be able to create sprites in one line instead of three.

def icon_maker(self, n, icon_x, icon_y):
    """
    Generates icons as sprites
    """
    sprite = displayio.TileGrid(self.sprite_sheet, pixel_shader=self.palette,
    width=1, height=1, tile_width=40, tile_height=40,
    default_tile=n, x=icon_x, y=icon_y)
    return sprite

_label_maker

_label_maker makes creating labels slightly more readable.

def _label_maker(self, text, x, y, font=None):
    """
    Generates labels
    """
    if not font:
        font = self.arial24
    return label.Label(font=font, x=x, y=y, text=text, color=self.WHITE)

_get_y

This uses the number of enabled devices, num_enabled to determine the placement of labels and sprites in the UI.

def _get_y(self):
    """
    Helper function for setup_display. Gets the y values used for sprites and labels.
    """
    enabled = self.num_enabled

    if self.heart_enabled:
        self._heart_y = 45* (self.num_enabled - enabled) + 75
        enabled -= 1
    if self.speed_enabled:
        self._speed_y = 45* (self.num_enabled - enabled) + 75
        enabled -= 1
    if self.cadence_enabled:
setup_display

This creates all the UI elements that do not change. First it creates a rectangle at the top of the screen to hold the heading, then it creates the heading. Then it places the sprites from the sprite sheet at locations that correspond with how many devices are enabled.

def setup_display(self):
    ""
    Prepares the display to show sensor values: Adds a header, a heading, and various sprites.
    ""
    self._get_y()
    sprites = displayio.Group()
    rect = Rect(0, 0, 240, 50, fill=self.PURPLE)
    self.splash.append(rect)
    heading = label.Label(font=self.arial24, x=55, y=25, text="Pyloton",
                          color=self.YELLOW)
    self.splash.append(heading)
    if self.heart_enabled:
        heart_sprite = self.icon_maker(0, 2, self._heart_y - 20)
        sprites.append(heart_sprite)
    if self.speed_enabled:
        speed_sprite = self.icon_maker(1, 2, self._speed_y - 20)
        sprites.append(speed_sprite)
    if self.cadence_enabled:
        cadence_sprite = self.icon_maker(2, 2, self._cadence_y - 20)
        sprites.append(cadence_sprite)
    if self.ams_enabled:
        ams_sprite = self.icon_maker(3, 2, self._ams_y - 20)
        sprites.append(ams_sprite)
    self.splash.append(sprites)
    self.display.show(self.splash)
    while self.loading_group:
        self.loading_group.pop()

update_display

update_display is used to display the most recent values from the sensors on the screen. If it is running for the first time, it will create the labels, and set _setup to True upon finishing. From then on, the text of the labels is edited, which increases the refresh rate.
def update_display(self):  # pylint: disable=too-many-branches
    """
    Updates the display to display the most recent values
    """
    if self.speed_enabled or self.cadence_enabled:
        speed, cadence = self.read_s_and_c()
    if self.heart_enabled:
        heart = self.read_heart()
        if not self._setup:
            self._hr_label = self._label_maker('{} bpm'.format(heart), 50,
            self._heart_y)
            self.splash.append(self._hr_label)
        else:
            self._hr_label.text = '{} bpm'.format(heart)
    if self.speed_enabled:
        if not self._setup:
            self._sp_label = self._label_maker('{} m/h'.format(speed), 50,
            self._speed_y)
            self.splash.append(self._sp_label)
        else:
            self._sp_label.text = '{} m/h'.format(speed)
    if self.cadence_enabled:
        if not self._setup:
            self._cadence_label = self._label_maker('{} rpm'.format(cadence),
            self._cadence_y)
            self.splash.append(self._cadence_label)
        else:
            self._cadence_label.text = '{} rpm'.format(cadence)
    if self.ams_enabled:
        ams = self.read_ams()
        if not self._setup:
            self._ams_label = self._label_maker('{}.'.format(ams), 50,
            self._ams_y,
            font=self.arial16)
            self.splash.append(self._ams_label)
        else:
            self._ams_label.text = '{}'.format(ams)
    self._setup = True

ams_remote

**ams_remote** uses the built-in buttons and capacitive touch pads to control media playing on an Apple device using AppleMediaServices. Button 'A' is on the left and button 'B' is on the right. The three capacitive touch pads are at the bottom and are labeled 0, 1, and 2.

def ams_remote(self):
    """
    Allows the 2 buttons and 3 capacitive touch pads in the CLUE to function as a media remote.
    """
    try:
        # Capacitive touch pad marked 0 goes to the previous track
        if self.clue.touch_0:
            self.ams.previous_track()
        time.sleep(0.25)
# Capacitive touch pad marked 1 toggles pause/play
if self.clue.touch_1:
    self.ams.toggle_play_pause()
    time.sleep(0.25)

# Capacitive touch pad marked 2 advances to the next track
if self.clue.touch_2:
    self.ams.next_track()
    time.sleep(0.25)

# If button B (on the right) is pressed, it increases the volume
if 'B' in self.clue.were_pressed:
    self.ams.volume_up()
    time.sleep(0.1)

# If button A (on the left) is pressed, the volume decreases
if 'A' in self.clue.were_pressed:
    self.ams.volume_down()
    time.sleep(0.1)
except (RuntimeError, UnsupportedCommand, AttributeError):
    return

# SPDX-FileCopyrightText: 2020 Eva Herrada for Adafruit Industries
# SPDX-License-Identifier: MIT

A library for completing the Pyloton bike computer learn guide utilizing the Adafruit CLUE.

import time
import adafruit_ble
from adafruit_ble.advertising.standard import ProvideServicesAdvertisement
from adafruit_ble.advertising.standard import SolicitServicesAdvertisement
import board
import digitalio
import displayio
import adafruit_imageload
from adafruit_ble_cycling_speed_and_cadence import CyclingSpeedAndCadenceService
from adafruit_ble_heart_rate import HeartRateService
from adafruit_bitmap_font import bitmap_font
from adafruit_display_shapes.rect import Rect
from adafruit_display_text import label
from adafruit_ble_apple_media import AppleMediaService
from adafruit_ble_apple_media import UnsupportedCommand
import gamepad
import touchio

class Clue:
    ""
    A very minimal version of the CLUE library.
    The library requires the use of many sensor-specific libraries this project doesn't use, and they were taking up a lot of RAM.
    ""
    def __init__(self):
        self._i2c = board.I2C()
        self._touches = [board.D0, board.D1, board.D2]
        self._touch_threshold_adjustment = 0
        self._a = digitalio.DigitalInOut(board.BUTTON_A)
        self._a.switch_to_input(pull=digitalio.Pull.UP)
        self._b = digitalio.DigitalInOut(board.BUTTON_B)
        self._b.switch_to_input(pull=digitalio.Pull.UP)
        self._gamepad = gamepad.GamePad(self._a, self._b)
@property
def were_pressed(self):
    """Returns a set of buttons that have been pressed since the last time
    were_pressed was run."
    ret = set()
pPressed = self._gamepad.get_pressed()
    for button, mask in (('A', 0x01), ('B', 0x02)):
        if mask & pressed:
            ret.add(button)
    return ret

@property
def button_a(self):
    """`True` when Button A is pressed. `False` if not.""
    return not self._a.value

@property
def button_b(self):
    """`True` when Button B is pressed. `False` if not.""
    return not self._b.value

def _touch(self, i):
    if not isinstance(self._touches[i], touchio.TouchIn):
        self._touches[i] = touchio.TouchIn(self._touches[i])
        self._touches[i].threshold += self._touch_threshold_adjustment
    return self._contacts[i].value

@property
def touch_0(self):
    """Returns True when capacitive touchpad 0 is currently being pressed."
    return self._touch(0)

@property
def touch_1(self):
    """Returns True when capacitive touchpad 1 is currently being pressed."
    return self._touch(1)

@property
def touch_2(self):
    """Returns True when capacitive touchpad 2 is currently being pressed."
    return self._touch(2)

class Pyloton:
    """Contains the various functions necessary for doing the Pyloton learn guide."
    #pylint: disable=too-many-instance-attributes
    YELLOW = 0xFCFF00
    PURPLE = 0x64337E
    WHITE = 0xFFFFFF
    clue = Clue()

    def __init__(self, ble, display, circ, heart=True, speed=True, cad=True,
                 ams=True, debug=False): #pylint: disable=too-many-arguments
        self.debug = debug
        self.ble = ble
        self.display = display
        self.circumference = circ
self.heart_enabled = heart
self.speed_enabled = speed
self.cadence_enabled = cad
self.ams_enabled = ams
self.hr_connection = None
self.num_enabled = heart + speed + cad + ams
self._previous_wheel = 0
self._previous_crank = 0
self._previous_revolutions = 0
self._previous_speed = 0
self._previous_cadence = 0
self._previous_heart = 0
self._speed_failed = 0
self._cadence_failed = 0
self._setup = 0
self._hr_label = None
self._sp_label = None
self._cadence_label = None
self._ams_label = None
self._hr_service = None
self._heart_y = None
self._speed_y = None
self._cadence_y = None
self._ams_y = None
self.ams = None
self.cyc_connections = None
self.cyc_services = None
self.track_artist = True

self.start = time.time()

self.splash = displayio.Group()
self.loading_group = displayio.Group()

self._load_fonts()

self.sprite_sheet, self.palette = adafruit_imageload.load("/sprite_sheet.bmp",
bitmap=displayio.Bitmap,
palette=displayio.Palette)

self.text_group = displayio.Group()

self.status = label.Label(font=self.arial12, x=10, y=200,
text='', color=self.YELLOW)
self.status1 = label.Label(font=self.arial12, x=10, y=220,
text='', color=self.YELLOW)

self.text_group.append(self.status)
self.text_group.append(self.status1)

def show_splash(self):
    """
    Shows the loading screen
    """
    if self.debug:
        return

    blinka_bitmap = "blinka-pyloton.bmp"

    # Compatible with CircuitPython 6 & 7
    with open(blinka_bitmap, 'rb') as bitmap_file:
        bitmap1 = displayio.OnDiskBitmap(bitmap_file)
tile_grid = displayio.TileGrid(bitmap1, pixel_shader=getattr(bitmap1, 'pixel_shader', displayio.ColorConverter()))
self.loading_group.append(tile_grid)
self.display.show(self.loading_group)
status_heading = label.Label(font=self.arial16, x=80, y=175,
text="Status", color=self.YELLOW)
rect = Rect(0, 165, 240, 75, fill=self.PURPLE)
self.loading_group.append(rect)
self.loading_group.append(status_heading)

def _load_fonts(self):
    
    Loads fonts
    
    self.arial12 = bitmap_font.load_font("/fonts/Arial-12.bdf")
self.arial16 = bitmap_font.load_font("/fonts/Arial-16.bdf")
self.arial24 = bitmap_font.load_font("/fonts/Arial-Bold-24.bdf")
glyphs = b'0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ-!,. ",\?!'
self.arial12.load_glyphs(glyphs)
self.arial16.load_glyphs(glyphs)
self.arial24.load_glyphs(glyphs)

def _status_update(self, message):
    
    Displays status updates
    
    if self.debug:
        print(message)
    return
    if self.text_group not in self.loading_group:
        self.loading_group.append(self.text_group)
    self.status.text = message[:25]
    self.status1.text = message[25:50]

def timeout(self):
    
    Displays Timeout on screen when pyloton has been searching for a sensor for too long
    
    self._status_update("Pyloton: Timeout")
time.sleep(3)

def heart_connect(self):
    
    Connects to heart rate sensor
    
    self._status_update("Heart Rate: Scanning...")
    for adv in self.ble.start_scan(ProvideServicesAdvertisement, timeout=5):
        if HeartRateService in adv.services:
            self._status_update("Heart Rate: Found an advertisement")
self.hr_connection = self.ble.connect(adv)
self._status_update("Heart Rate: Connected")
    break
self.ble.stop_scan()
if self.hr_connection:
    self.hr_service = self.hr_connection[HeartRateService]
return self.hr_connection

@staticmethod
def _has_timed_out(start, timeout):
    if time.time() - start >= timeout:
        return True
    return False

def ams_connect(self, start=time.time(), timeout=30):
    """
    Connect to an Apple device using the ble_apple_media library
    """
    self.status_update("AppleMediaService: Connect your phone now")
    radio = adafruit_ble.BLERadio()
    a = SolicitServicesAdvertisement()
    a.solicited_services.append(AppleMediaService)
    radio.start_advertising(a)
    while not radio.connected and not self._has_timed_out(start, timeout):
        pass
    self.status_update("AppleMediaService: Connected")
    for connection in radio.connections:
        if not connection.paired:
            connection.pair()
        self.status_update("AppleMediaService: Paired")
    self.ams = connection[AppleMediaService]
    return radio

def speed_cadence_connect(self):
    """
    Connects to speed and cadence sensor
    """
    self.status_update("Speed and Cadence: Scanning...")
    # Save advertisements, indexed by address
    advs = {}
    for adv in self.ble.start_scan(ProvideServicesAdvertisement, timeout=5):
        if CyclingSpeedAndCadenceService in adv.services:
            self.status_update("Speed and Cadence: Found an advertisement")
            # Save advertisement. Overwrite duplicates from same address
            advs[adv.address] = adv
    self.ble.stop_scan()
    self.status_update("Speed and Cadence: Stopped scanning")
    if not advs:
        # Nothing found. Go back and keep looking.
        return []
    # Connect to all available CSC sensors.
    self.cyc_connections = []
    for adv in advs.values():
        self.cyc_connections.append(self.ble.connect(adv))
        self.status_update("Speed and Cadence: Connected {}".format(len(self.cyc_connections)))
    self.cyc_services = []
    for conn in self.cyc_connections:
        self.cyc_services.append(conn[CyclingSpeedAndCadenceService])
    self.status_update("Pyloton: Finishing up...")
    return self.cyc_connections
def _compute_speed(self, values, speed):
    wheel_diff = values.last_wheel_event_time - self._previous_wheel
    rev_diff = values.cumulative_wheel_revolutions - self._previous_revolutions
    if wheel_diff:
        # Rotations per minute is 60 times the amount of revolutions since
        # the last update over the time since the last update
        rpm = 60*(rev_diff/(wheel_diff/1024))
        # We then multiply it by the wheel's circumference and convert it to mph
        speed = round((rpm * self.circumference) * (60/63360), 1)
        if speed < 0:
            speed = self._previous_speed
            self._previous_speed = speed
            self._previous_revolutions = values.cumulative_wheel_revolutions
            self._speed_failed = 0
        else:
            self._speed_failed += 1
            if self._speed_failed >= 3:
                speed = 0
    self._previous_wheel = values.last_wheel_event_time
    self._previous_revolutions = values.cumulative_wheel_revolutions
    return speed

def _compute_cadence(self, values, cadence):
    crank_diff = values.last_crank_event_time - self._previous_crank
    crank_rev_diff = values.cumulative_crank_revolutions - self._previous_rev
    if crank_rev_diff:
        # Rotations per minute is 60 times the amount of revolutions since the
        # last update over the time since the last update
        cadence = round(60*(crank_rev_diff/(crank_diff/1024)), 1)
        if cadence < 0:
            cadence = self._previous_cadence
            self._previous_cadence = cadence
            self._previous_rev = values.cumulative_crank_revolutions
            self._cadence_failed = 0
        else:
            self._cadence_failed += 1
            if self._cadence_failed >= 3:
                cadence = 0
    self._previous_crank = values.last_crank_event_time
    return cadence

def read_s_and_c(self):
    """Reads data from the speed and cadence sensor
    ""
    speed = self._previous_speed
    cadence = self._previous_cadence
    for conn, svc in zip(self.cyc_connections, self.cyc_services):
        if not conn.connected:
            speed = cadence = 0
            continue
        values = svc.measurement_values
        if not values:
            if self._cadence_failed >= 3 or self._speed_failed >= 3:
                if self._cadence_failed > 3:
                    cadence = 0
                if self._speed_failed > 3:
                    speed = 0
            continue
            if not values.last_wheel_event_time:
                continue
            speed = self._compute_speed(values, speed)
            if not values.last_crank_event_time:
                continue
cadence = self._compute_cadence(values, cadence)

if speed:
    speed = str(speed)[:8]
if cadence:
    cadence = str(cadence)[:8]
return speed, cadence

def read_heart(self):
    """
    Reads date from the heart rate sensor
    """
    measurement = self._hr_service.measurement_values
    if measurement is None:
        heart = self._previous_heart
    else:
        heart = measurement.heart_rate
        self._previous_heart = measurement.heart_rate
    if heart:
        heart = str(heart)[:4]
    return heart

def read_ams(self):
    """
    Reads data from AppleMediaServices
    """
    current = time.time()
    try:
        if current - self.start > 3:
            self.track_artist = not self.track_artist
            self.start = time.time()
        if self.track_artist:
            data = self.ams.artist
        if not self.track_artist:
            data = self.ams.title
    except (RuntimeError, UnicodeError):
        data = None
    if data:
        data = data[:16] + (data[16:] and '..')
    return data

def icon_maker(self, n, icon_x, icon_y):
    """
    Generates icons as sprites
    """
    sprite = displayio.TileGrid(self.sprite_sheet, pixel_shader=self.palette,
                                 width=1, height=1, tile_width=40, tile_height=40,
                                 default_tile=n,
                                 x=icon_x, y=icon_y)
    return sprite

def _label_maker(self, text, x, y, font=None):
    """
    Generates labels
    """
    if not font:
        font = self.arial24
    return label.Label(font=font, x=x, y=y, text=text, color=self.WHITE)
def _get_y(self):
    """
    Helper function for setup_display. Gets the y values used for sprites and
    labels.
    """
    enabled = self.num_enabled

    if self.heart_enabled:
        self._heart_y = 45*(self.num_enabled - enabled) + 75
        enabled -= 1
    if self.speed_enabled:
        self._speed_y = 45*(self.num_enabled - enabled) + 75
        enabled -= 1
    if self.cadence_enabled:
        self._cadence_y = 45*(self.num_enabled - enabled) + 75
        enabled -= 1
    if self.ams_enabled:
        self._ams_y = 45*(self.num_enabled - enabled) + 75
        enabled -= 1

def setup_display(self):
    """
    Prepares the display to show sensor values: Adds a header, a heading, and
    various sprites.
    """
    self._get_y()
    sprites = displayio.Group()

    rect = Rect(0, 0, 240, 50, fill=self.PURPLE)
    self.splash.append(rect)

    heading = label.Label(font=self.arial24, x=55, y=25, text="Pyloton",
                          color=self.YELLOW)
    self.splash.append(heading)

    if self.heart_enabled:
        heart_sprite = self.icon_maker(0, 2, self._heart_y - 20)
        sprites.append(heart_sprite)
    if self.speed_enabled:
        speed_sprite = self.icon_maker(1, 2, self._speed_y - 20)
        sprites.append(speed_sprite)
    if self.cadence_enabled:
        cadence_sprite = self.icon_maker(2, 2, self._cadence_y - 20)
        sprites.append(cadence_sprite)
    if self.ams_enabled:
        ams_sprite = self.icon_maker(3, 2, self._ams_y - 20)
        sprites.append(ams_sprite)

    self.splash.append(sprites)
    self.display.show(self.splash)
    while self.loading_group:
        self.loading_group.pop()

def update_display(self): #pylint: disable=too-many-branches
    """
    Updates the display to display the most recent values
    """
    if self.speed_enabled or self.cadence_enabled:
        speed, cadence = self.read_s_and_c()
    if self.heart_enabled:
        heart = self.read_heart()
if not self._setup:
    self._hr_label = self._label_maker('{} bpm'.format(heart), 50, self._heart_y) # 75
    self.splash.append(self._hr_label)
else:
    self._hr_label.text = '{} bpm'.format(heart)

if self.speed_enabled:
    if not self._setup:
        self._sp_label = self._label_maker('{} mph'.format(speed), 50, self._speed_y) # 120
        self.splash.append(self._sp_label)
    else:
        self._sp_label.text = '{} mph'.format(speed)

if self.cadence_enabled:
    if not self._setup:
        self._cadence_label = self._label_maker('{} rpm'.format(cadence), 50, self._cadence_y)
        self.splash.append(self._cadence_label)
    else:
        self._cadence_label.text = '{} rpm'.format(cadence)

if self.ams_enabled:
    ams = self.read_ams()
    if not self._setup:
        self._ams_label = self._label_maker('{}'.format(ams), 50, self._ams_y, font=self.arial16)
        self.splash.append(self._ams_label)
    else:
        self._ams_label.text = '{}'.format(ams)

self._setup = True

def ams_remote(self):
    """
    Allows the 2 buttons and 3 capacitive touch pads in the CLUE to function as
    a media remote.
    """
    try:
        # Capacitive touch pad marked 0 goes to the previous track
        if self.clue.touch_0:
            self.ams.previous_track()
            time.sleep(0.25)

        # Capacitive touch pad marked 1 toggles pause/play
        if self.clue.touch_1:
            self.ams.toggle_play_pause()
            time.sleep(0.25)

        # Capacitive touch pad marked 2 advances to the next track
        if self.clue.touch_2:
            self.ams.next_track()
            time.sleep(0.25)

        # If button B (on the right) is pressed, it increases the volume
        if self.clue.button_b:
            self.ams.volume_up()
            time.sleep(0.1)

        # If button A (on the left) is pressed, the volume decreases
        if self.clue.button_a:
            self.ams.volume_down()
            time.sleep(0.1)
    except (RuntimeError, UnsupportedCommand, AttributeError):
        return
3D Printing

First, you'll need to 3D print the Pyloton case and lid, as well as a mounting bracket or wrist mounting bracelet if you want to use that version.

For bike mounting, there are a couple of options of bracket size, depending on the diameter of your handlebars.

3D Printed Parts

STL files for 3D printing are oriented to print "as-is" on FDM style machines. Original design source may be downloaded using the links below.

- clue-frame.stl
- clue-bike-case-btns.stl
- clue-bike-lid-tripod.stl or clue-bike-lid-5m.stl
- clue-bike-bracket-tri-*.stl or clue-bike-bracket-m5-*.stl
  - 24mm or 31mm diameter
Slicing Parts

Supports recommended around the lanyard ears and slide switch. Slice with setting for PLA material.

The parts were sliced using CURA using the slice settings below.

- PLA filament 210c extruder
- 0.2 layer height
- 10% gyroid infill
- 60mm/s print speed
- 70c heated bed

Supports - Place supports under the slide switch walls as shown

- STLs CLUE bike mount (31mm or 24mm bracket M5) and (1/4-20 case and brackets)
- Edit CLUE bike mount (24mm diameter 1/4 - 20 mount)
- Edit bracket (31.8mm M5 screws)
- Edit bracket (24mm diameter M5)
- Edit M5 lid

For full instructions on how to assemble and customize flexible watch bands, check out the wearable case guide: [https://learn.adafruit.com/clue-case/3d-printing](https://learn.adafruit.com/clue-case/3d-printing)

Full instructions for Wearable case
In order to power the Pyloton, we'll need to create a switching battery extension, as shown in this excellent guide, [DIY On/Off JST Switch Adapter](#).

---

**Use the Pyloton**
Time to use the Pyloton and get your quantified self out on a bike ride!

Make sure you've got your heart rate monitor turned on and strapped to your wrist or chest, and give your bike crank and wheel a little wiggle to make sure the sensors are awake.

Then, turn on the Pyloton. It'll begin looking for connections to the iOS device, heart rate monitor, and cadence & speed sensors. The first one it will attempt is the connection to the iOS device.

iOS Connection

With Bluetooth turned on on the iOS device, you will see the Pyloton pop up in the Other Devices list. Go ahead and click it.

Note: the device may have a name like CIRCUITPYadde as seen here. Pick the device and it will make pop up the Bluetooth Pairing Request dialog box. Click Pair.

The Pyloton will show up in your My Devices list as Connected, and it will automatically connect in the future.
AMS Connection
The Pyloton will display that it is making a connection to the Apple Media Service.
Sensor Connections

Next, the Pyloton will look for and connect to a heart rate monitor, followed by a cadence & speed sensor. There is no pairing/bonding step for these, the Pyloton will simply connect to the first ones it encounters that are advertising those services.
Display
The Pyloton will now display:

- Heart rate
- Cycling Speed
- Cycling Cadence
- Song title and artist track info

Get Track Info

Go ahead and launch a media player app, such as Spotify.

You'll see that the Pyloton displays the track title and artist (alternating every few seconds), in the track info box.

Change the song on your iOS device, and it will update on the Pyloton.

Send Media Control Commands
You can also send the player commands from the Pyloton.

- Press the B button (on the right) of the CLUE to lower the volume or the A button to increase it.

- The three capacitive touch sensors on the edge connector of the CLUE are used for more media controls:
  - Pad 0 selects the previous track
  - Pad 1 pauses/plays the current track
  - Pad 2 selects the next track