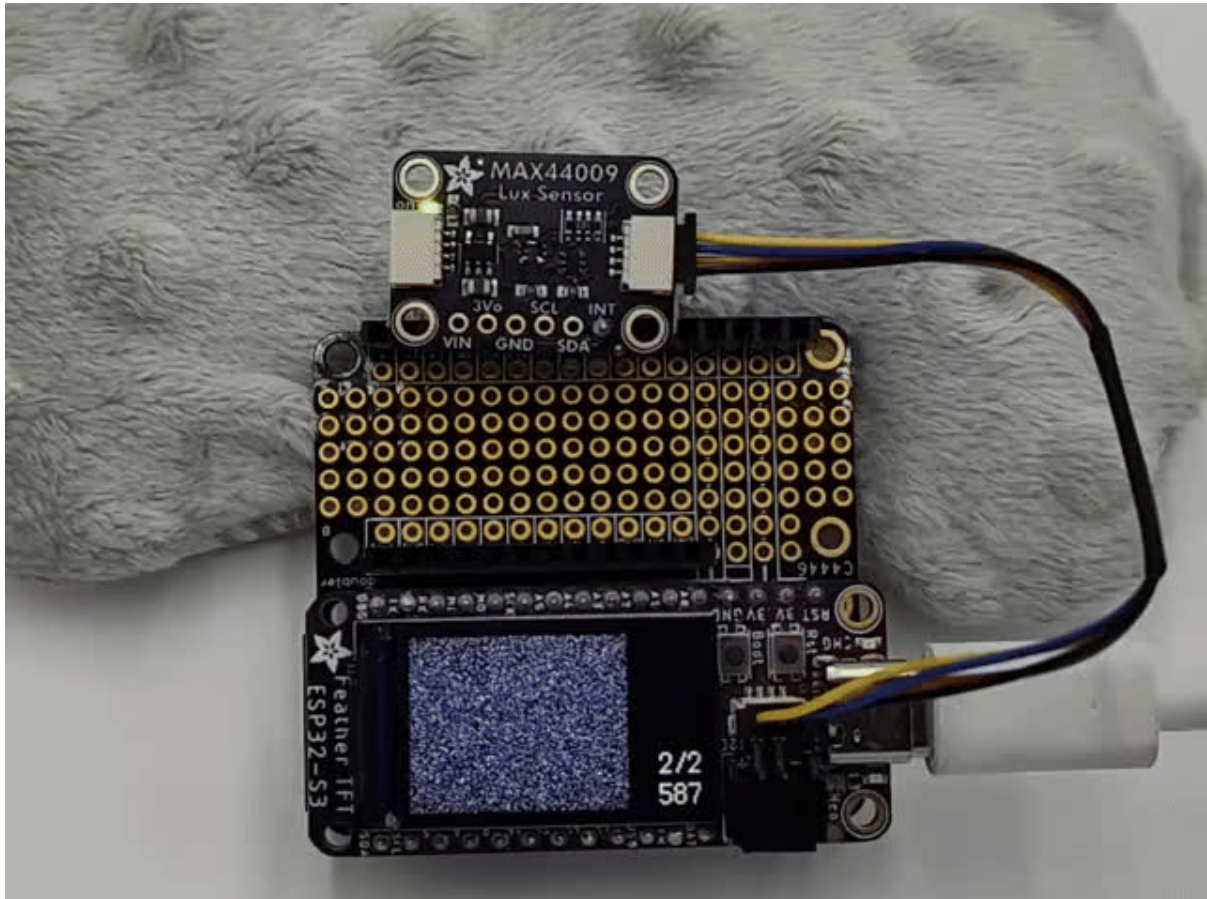




Sensor-Locked Secrets with CircuitPython

Created by Tim C



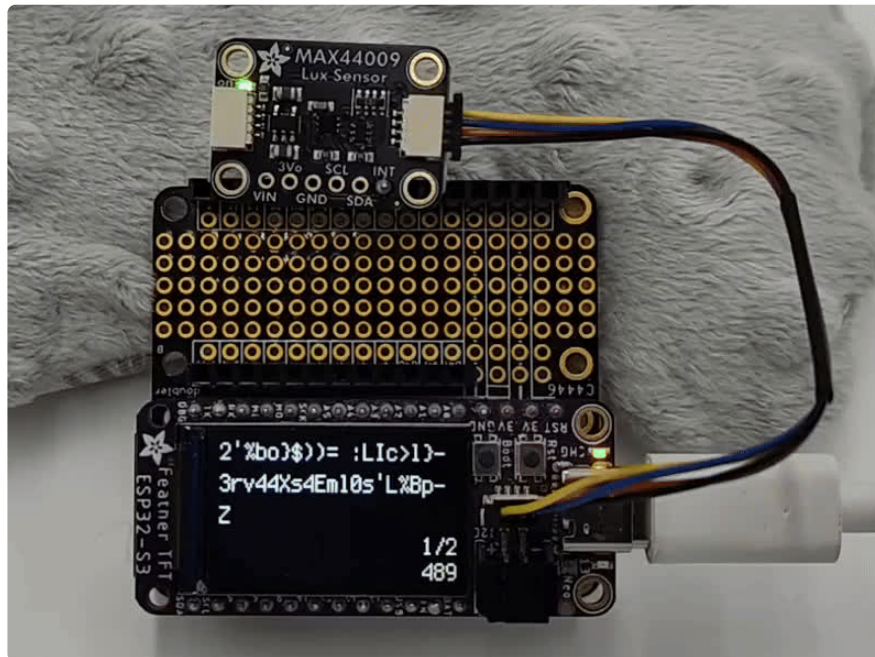
<https://learn.adafruit.com/sensor-locked-secrets-with-circuitpython>

Last updated on 2026-05-12 01:48:44 PM UTC

Table of Contents

Overview	3
• Parts	
Installing CircuitPython	5
• CircuitPython Quickstart	
How It Works	8
• Text Secrets	
• Image File Secrets	
Encrypting Secrets	9
• Text Secrets	
• Image Secrets	
GPS Coordinates	12
• Hardware	
• Encrypt Secrets	
• Code	
• Drive Structure	
Lux Level	23
• Hardware	
• Encrypt Secrets	
• Code	
• Drive Structure	
Temperature Humidity CO2	35
• Hardware	
• Encrypt Secrets	
• Code	
• Drive Structure	
Using Other Sensors	47
• Different Types of Sensors	

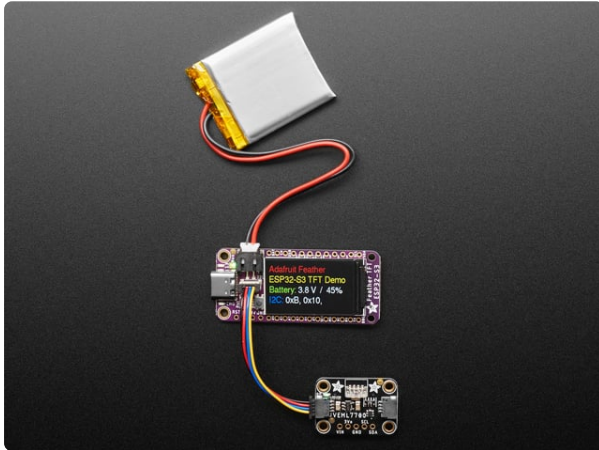
Overview



This project is about scrambling a secret message or picture in such a way that it can only be decrypted when certain environmental factors are met. The guide covers three different types of sensor inputs: GPS coordinates, lux light values, and temperature/humidity/CO2 readings. The general concept could be adapted for other types of sensors like magnetic, gyroscope, proximity and more.

The inspirations for this project include [reverse geo-cache puzzles \(https://adafru.it/1aDn\)](https://adafru.it/1aDn) like the one in [this learn guide \(https://adafru.it/1aDo\)](https://adafru.it/1aDo), and statues that are engineered to use the sun's location at specific day and time to create a special effect with shadows or sun beams. A few historical examples of the latter are ‘the descent of Kukulcán’ serpent effect on the Kukulcán pyramid in Chichén Itzá Mexico, and the [inner chamber illumination at Newgrange \(https://adafru.it/1aDp\)](https://adafru.it/1aDp) in County Meath Ireland.

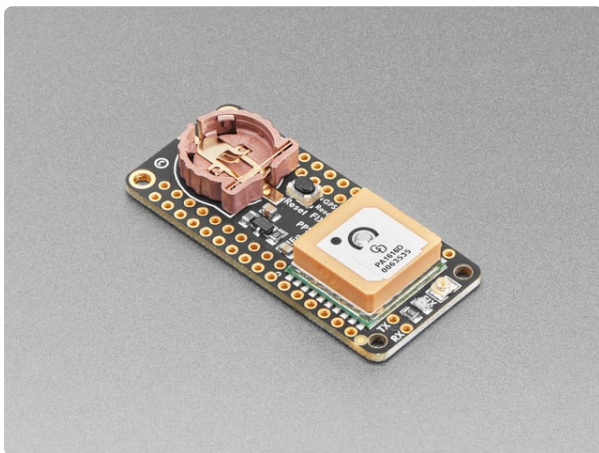
Parts



[Adafruit ESP32-S3 TFT Feather - 4MB Flash, 2MB PSRAM, STEMMA QT](https://www.adafruit.com/product/5483)

We've got a new machine here at Adafruit, it can uncover your deepest desires. Don't believe me? I'll turn it on right now to prove it to you! What, you want your very own...

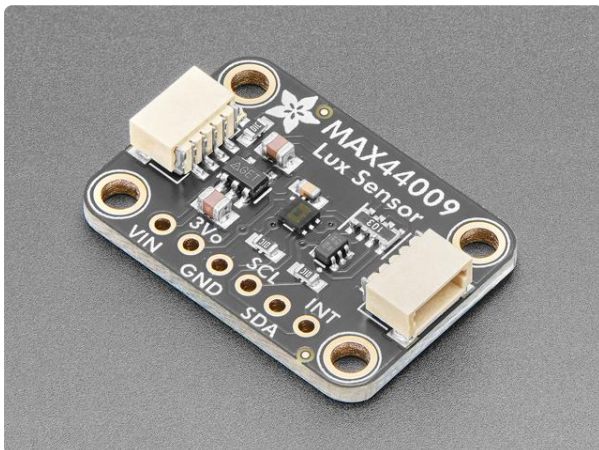
<https://www.adafruit.com/product/5483>



[Adafruit Ultimate GPS FeatherWing](https://www.adafruit.com/product/3133)

Give your Feather a sense of place, with an Ultimate GPS FeatherWing. In 2013 we designed the Ultimate GPS...

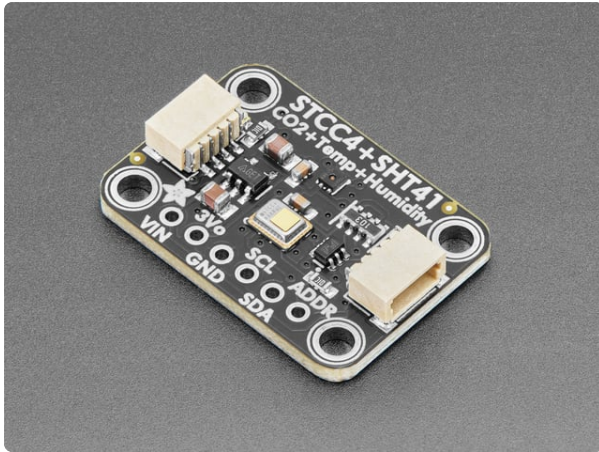
<https://www.adafruit.com/product/3133>



[Adafruit MAX44009 Wide-range Lux Light Sensor - 188,000 Lux Max](https://www.adafruit.com/product/6498)

This is an easy to use lux sensor, and features an ultra-wide 22-bit dynamic range from 0.045lux to 188,000 lux. That means you can use the Adafruit MAX44009...

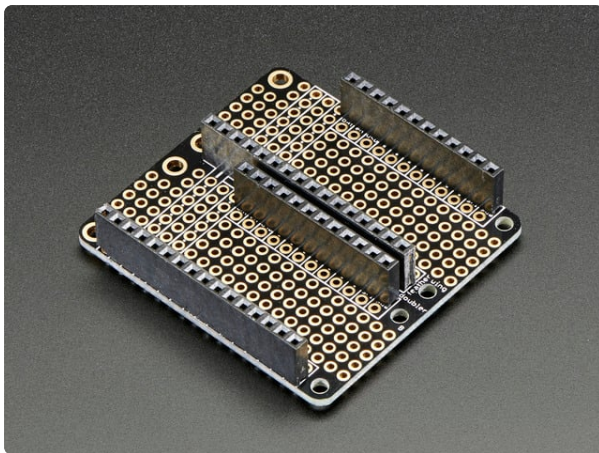
<https://www.adafruit.com/product/6498>



[Adafruit STCC4 and SHT41 - CO2, Temperature & Humidity Sensor](https://www.adafruit.com/product/6478)

CO₂ sensors are essential for determining if a room is too 'stuffy' - high CO₂ makes humans grumpy and tired. That's why it's always nice to take a deep breath...

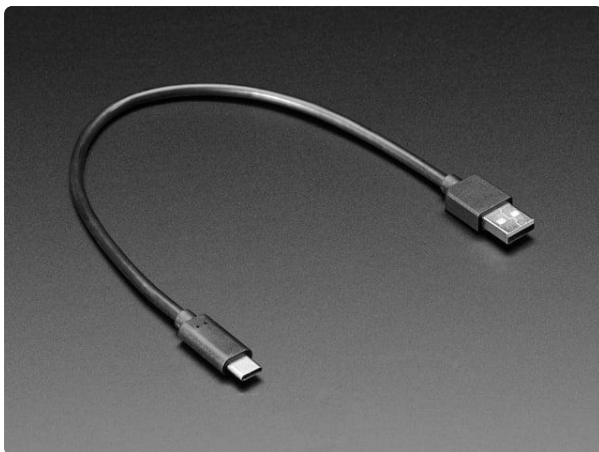
<https://www.adafruit.com/product/6478>



[FeatherWing Doubler - Prototyping Add-on For All Feather Boards](https://www.adafruit.com/product/2890)

This is the FeatherWing Doubler - a prototyping add-on and more for all Feather boards. This is similar to our

<https://www.adafruit.com/product/2890>



[USB Type A to Type C Cable - 1ft - 0.3 meter](https://www.adafruit.com/product/4473)

As technology changes and adapts, so does Adafruit. This USB Type A to Type C cable will help you with the transition to USB C, even if you're still...

<https://www.adafruit.com/product/4473>

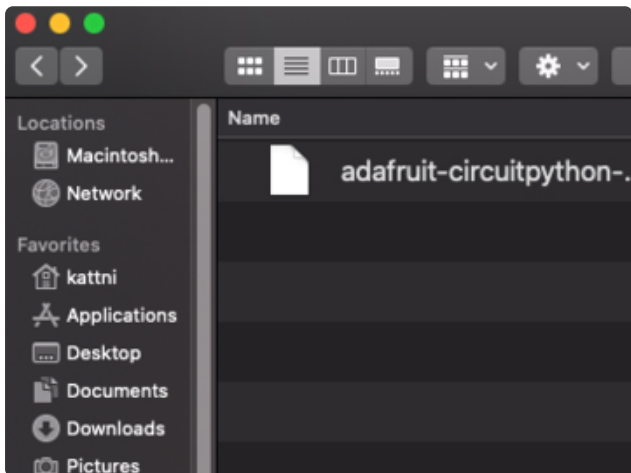
Installing CircuitPython

[CircuitPython](https://adafru.it/tB7) (<https://adafru.it/tB7>) is a derivative of [MicroPython](https://adafru.it/BeZ) (<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. Simply copy and edit files on the **CIRCUITPY** drive to iterate.

CircuitPython Quickstart

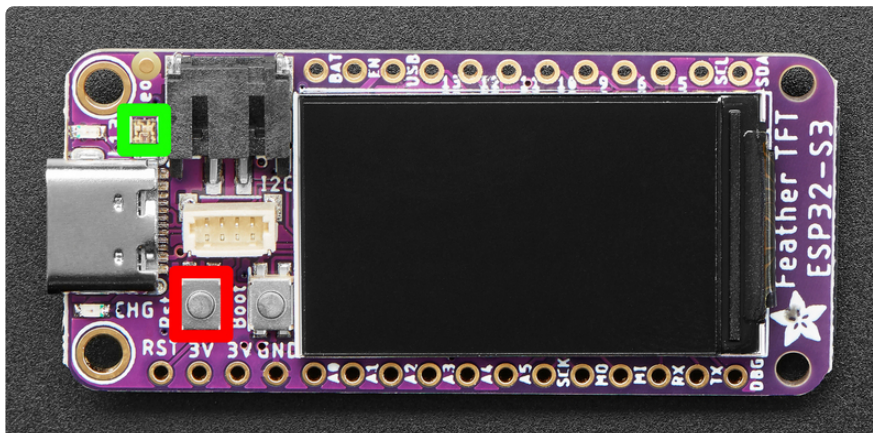
Follow this step-by-step to quickly get CircuitPython running on your board.

<https://adafru.it/10yc>



Click the link above to download the latest CircuitPython UF2 file.

Save it wherever is convenient for you.



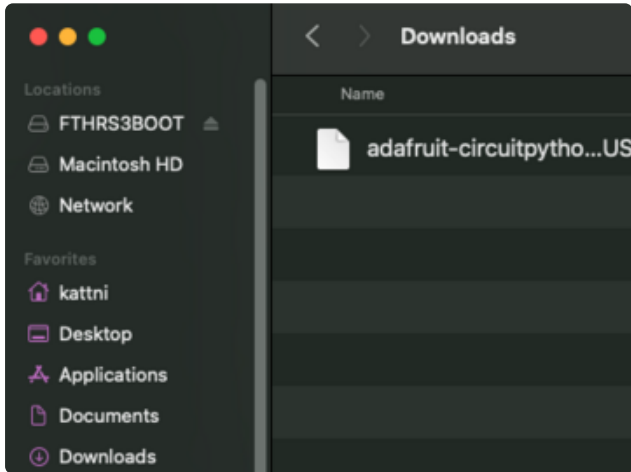
Plug your board into your computer, using a known-good data-sync cable, directly, or via an adapter if needed.

Double-click the **reset** button (highlighted in red above), and you will see the **RGB status LED(s)** turn green (highlighted in green above). If you see red, try another port, or if you're using an adapter or hub, try without the hub, or different adapter or hub.

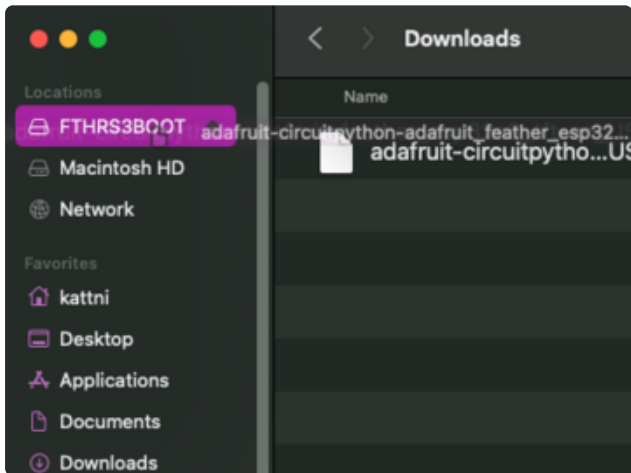
For this board, tap reset and wait for the **LED** to turn purple, and as soon as it turns purple, tap reset again. The second tap needs to happen while the LED is still purple.

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

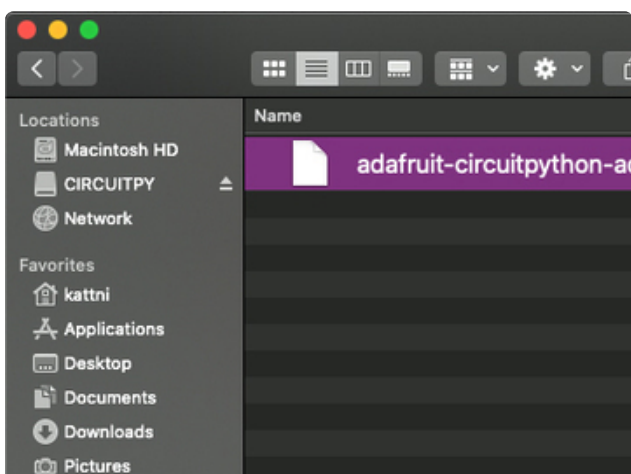
A lot of people end up using charge-only USB cables and it is very frustrating! **Make** sure you have a **USB** cable you know is good for data sync.



You will see a new disk drive appear called **FTHRS3BOOT**.



Drag the **adafruit_circuitpython_etc.uf2** file to **FTHRS3BOOT**.



The **BOOT** drive will disappear and a new disk drive called **CIRCUITPY** will appear.

That's it!

How It Works

The project supports encrypting a text message or an image file. The cipher used differs between them, but the key derivation works similarly. The key is generated by finding the [SHA-256 hash](https://adafru.it/1aDq) (https://adafru.it/1aDq) of the sensor reading range (ex: `200-300`) or GPS coordinates (ex: `40.656, -74.007`).

Text Secrets

Text messages use the [Vigenère cipher](https://adafru.it/1aDr) (https://adafru.it/1aDr) modified to support the full range of printable ASCII characters instead of just letters A-Z. This cipher outputs scrambled, but printable, text when an attempt to decrypt with the incorrect key is made. It's perfect for this project because it shows mysterious text as a clue that a secret message is hidden within. Vigenère cipher can use a key of any length. The key must use the same alphabet of characters that the encrypted message does. In this case, the ASCII printable characters in the ID range 32-126 from **space** to **tilde** in the [ASCII chart](https://adafru.it/1aDs) (https://adafru.it/1aDs). SHA-256 produces a 256-bit (32 byte) hash containing raw byte values which aren't restricted to printable ASCII. To work around this, the base64 representation of the hash is used as the Vigenère key. For example, a generated key looks something like this:

```
PdToaSE3R9V9C1RENwNMhYV+ihPv5NXt05kQo8VrbH8=
```

Once you have the key, encrypting a message with the Vigenère cipher is done by looking up each clear text characters cipher text counterpart in the Vigenère table or [tabula recta](https://adafru.it/1aDt) (https://adafru.it/1aDt). Each successive character that is encoded uses a row from the table matching the next character from the key, which gives the cipher its [polyalphabetic](https://adafru.it/1aDu) (https://adafru.it/1aDu) property. That increases its resilience against basic frequency analysis, and makes repeated clear text characters come out looking scrambled and non-repetitive in the cipher text.

Image File Secrets

Image files contain raw binary data instead of only printable text characters. Modern cryptography ciphers were made for the computer age and support encrypting raw data. [AES](https://adafru.it/Nap) (https://adafru.it/Nap) is one such modern cipher. It's used by this project for encrypting images. Under normal circumstances, an entire file would be encrypted with AES, making it completely unreadable and unable to be successfully rendered as an image in common applications.

The image file contains metadata like encoding type, size, and color palette which are necessary for rendering it. To avoid completely non-functional image files, only the pixel data is encrypted. The rest of the metadata remains intact. As a result, the display will show a scrambled image that resembles TV static whenever the key is wrong, instead of being entirely broken with only an error message about a corrupt file.

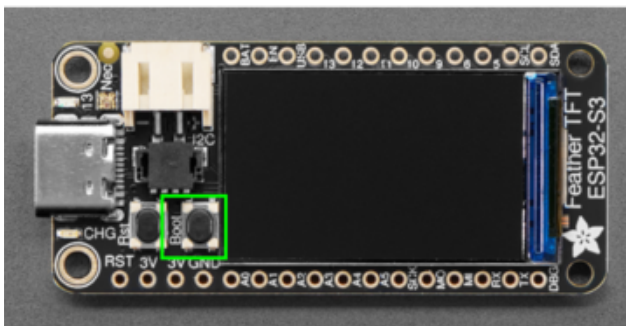
One of the variations of AES accepts a 256-bit (32 byte) key, which is conveniently exactly what is output by the SHA-256 hashing algorithm. That means the raw bytes of the hash can be used as the key directly instead of having to convert them to base64 or any other representation. Here is an example of what the AES key could look like:

```
b'9\x0b6|\xf2\xc3Cg\xca\xb5}  
I3,m\xcb\x0f\x8e\xb7\xb1\x08J` \xcb\x84\xd1]E\xeb\xe4\xe4\x8a'
```

The code attempts to decrypt the pixel data using a key derived from the current sensor reading, then uses [bitmaptools.arrayblit\(\)](https://adafru.it/1ayG) (<https://adafru.it/1ayG>) to copy the resulting pixels into a `displayio.Bitmap` so that it can be shown on the display. If the key was incorrect, the Bitmap will look like random pixel noise. If the key was correct, the original image will be revealed.

Encrypting Secrets

The different versions of CircuitPython microcontroller code for this project support different sensor breakouts for decoding the secret messages. The same encryptor page can be used to encode secrets for all of them.



The page can be used to encrypt multiple messages/images with different values. Each encrypted secret gets copied into a `SEQUENCE` list inside of the `code.py` file.

Once a secret has been revealed successfully, it remains visibly decrypted on the screen until the user presses the **BOOT** button on the Feather to advance to the next puzzle in the sequence.

Click the button bellow access the encryptor page.

<https://adafru.it/1aDv>

The page contains two tabs: one for encoding text, the other for encoding images.

Text Secrets

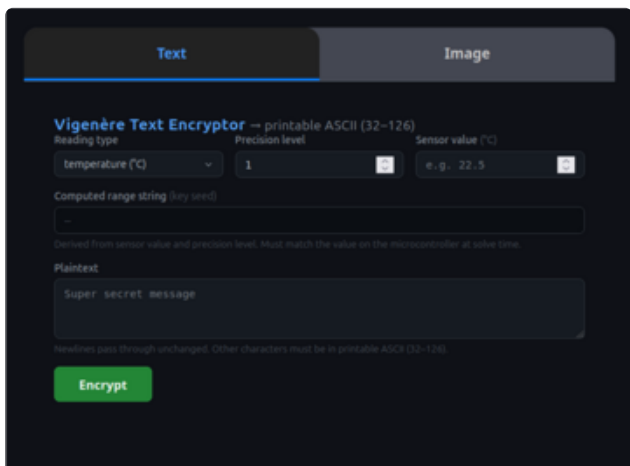
For text secrets, fill in the following fields and then press the **Encrypt** button.

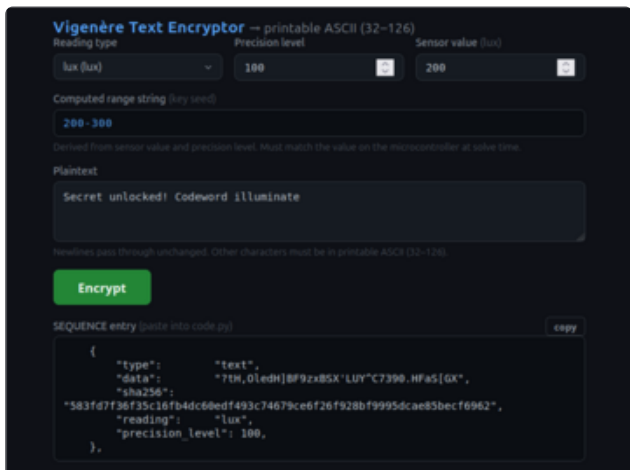
Reading type - a dropdown. Select **Temperature, Humidity, CO2, Lux, or Other** based on the type of sensor/data reading you want to unlock the secret.

Precision level - an integer number. Larger numbers result in a wider range of successful decryption values, **1** means the value must fall between two consecutive values i.e. **22-23**. Whereas **10** would mean a range of 10 values like **42-52**. For GPS, the precision level is used for the number of decimal places in the coordinates i.e. **40.656, -74.007** is **3** decimals.

Sensor value - a float number. The target sensor reading for unlocking the secret. The actual key will be derived from a range that covers the target value based on the precision level.

Plaintext - a text string input. The secret message that you wish to encrypt.





When you press **Encrypt**, a sequence entry text box will appear with an object containing the encrypted message along with metadata required to decrypt and validate it.

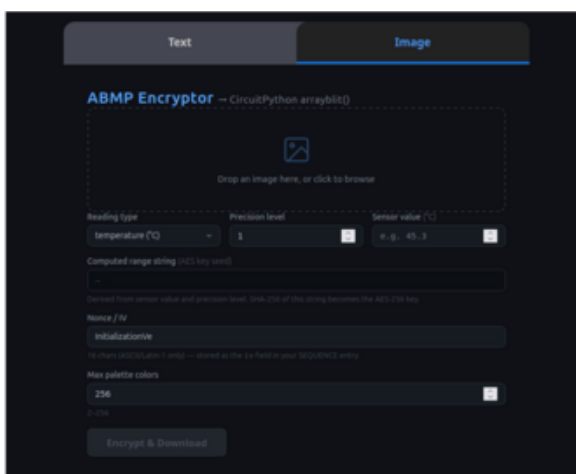
Press the **copy** button, along the top right of the box, and then paste the whole thing into the **SEQUENCE** list inside of **code.py** in the user configuration section near the top.

```

1 SEQUENCE = [
2   {
3     "type": "text",
4     "data": "?tH,OledH]BF9zxBSX'LUY^C7390.HFaS[GX",
5     "sha256": "583fd7f36f35c16fb4dc60edf493c74679ce6f26f928bf9995dcae85becf6962",
6     "reading": "lux",
7     "precision_level": 100,
8   },
9   # Add more entries here...
10 ]

```

Image Secrets



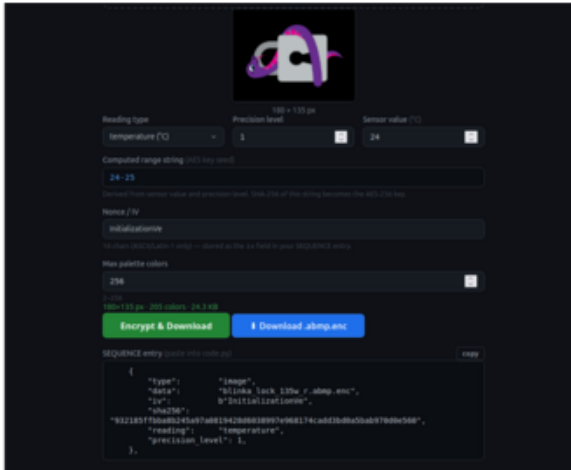
Encrypting image secrets is similar to text. The main differences are:

Drag and drop an image file on the page instead of entering plaintext into a box. The page supports PNG, JPEG, and BMP formatted images.

Nonce / IV - required for AES encryption. Leave the default or change it to any 16 character string.

Max palette colors - how many colors to use for the palette of the resulting Bitmap image. Using the default **256** is fine for most cases.

After you fill in the fields and click **Encrypt & Download** two things will happen. An encrypted copy of the image will be downloaded with a name like **some_image.abmp.enc**, and the sequence entry box will appear and get populated with an entry for the encoded image.



Press the **copy** button along the top right of the box and then paste the whole thing into the **SEQUENCE** list inside of **code.py** in the user configuration section near the top.

Make sure that the **data** value in the object has the exact name of the downloaded image file. You can change the name to anything, but the name of the file must match the **data** value in this object. Change them both together if you do.

Copy the encoded image file to the **CIRCUITPY** drive.

```

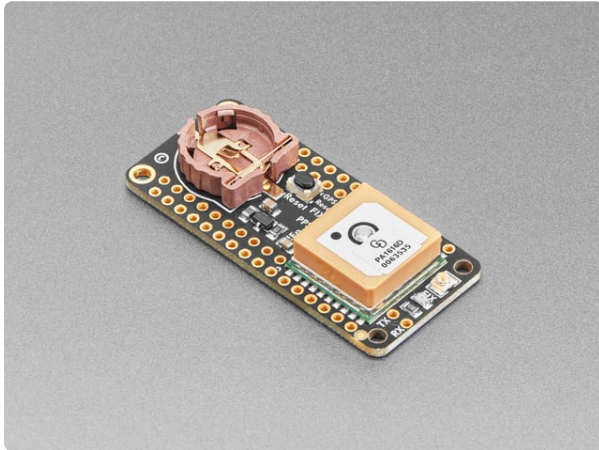
1 SEQUENCE = [
2   {
3     "type":    "image",
4     "data":    "secret_image.abmp.enc",
5     "iv":      b"InitializationVe",
6     "sha256":  "932185ffbba8b245a97a0819428d6038997e968174cadd3bd0a5bab970d0e560",
7     "reading": "temperature",
8     "precision_level": 1,
9   },
10  # Add more entries here...
11 ]

```

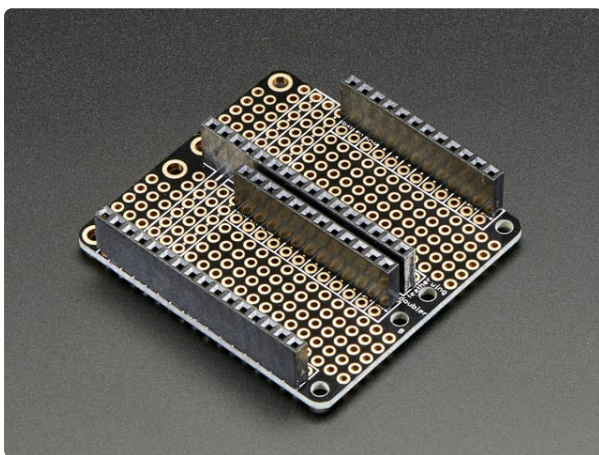
GPS Coordinates

This version of project code is basically a [reverse geo-cache puzzle](https://adafru.it/1aDw) like the ones that inspired the project. The key to encrypt secret text or image is derived from GPS coordinates. The secret can only be revealed by taking the device and GPS receiver to the specified location.

Hardware

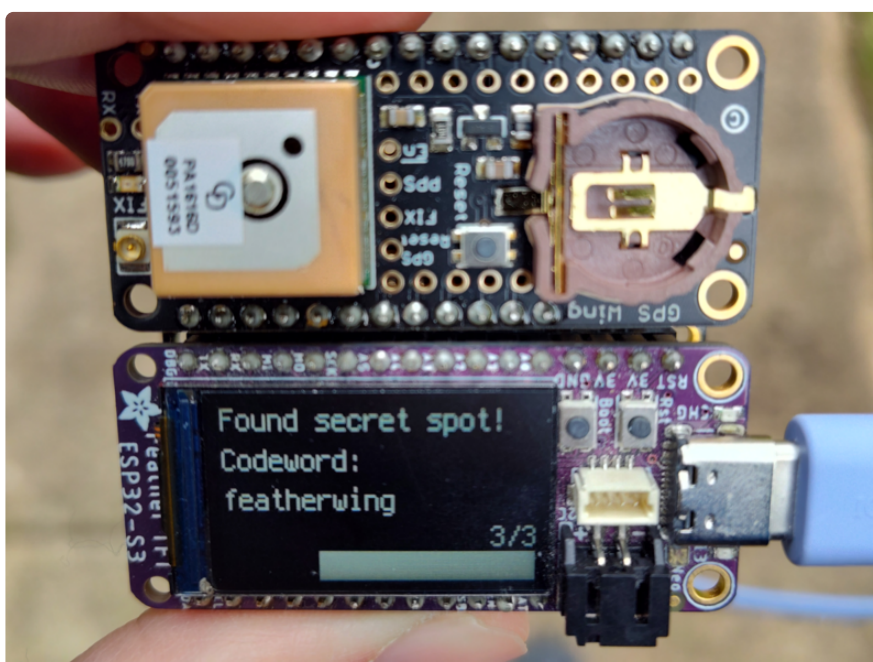


[Adafruit Ultimate GPS FeatherWing](https://www.adafruit.com/product/3133)
Give your Feather a sense of place, with an Ultimate GPS FeatherWing. In 2013 we designed the Ultimate GPS...
<https://www.adafruit.com/product/3133>



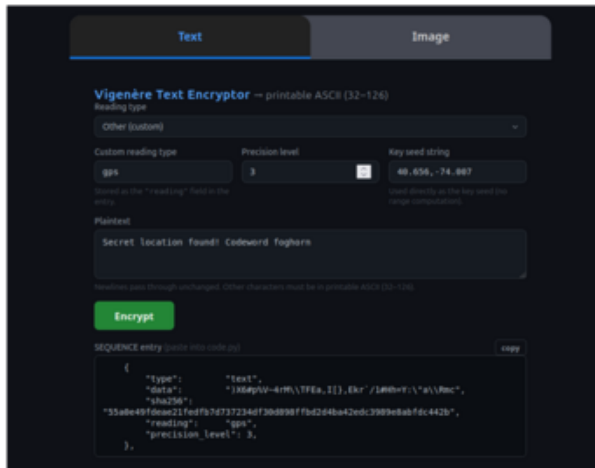
[FeatherWing Doubler - Prototyping Add-on For All Feather Boards](https://www.adafruit.com/product/2890)
This is the FeatherWing Doubler - a prototyping add-on and more for all Feather boards. This is similar to our
<https://www.adafruit.com/product/2890>

Connect the Feather S3 TFT to the GPS Featherwing using a FeatherWing doubler.



Encrypt Secrets

<https://adafru.it/1aDv>



To encrypt secrets for GPS coordinate unlocking use, the **Other (custom)** reading type value in the dropdown on the encryptor page. Use **gps** for the custom reading type and enter your coordinates in the key seed string field.

Ensure that your coordinates have the same number of decimal places that the precision level is set to, i.e. **3 decimals** equals **3** precision level.

Also make sure the coordinates are separated by a comma and that there are no spaces around the comma or anywhere else in the string. Example of correct syntax for precision level 3:

40.656, -74.007

Code

To use the application, you need to obtain **code.py** with the program, and the other project files to place on the Feather **CIRCUITPY** drive.

Thankfully, this can be done in one go. In the example below, click the **Download Project Bundle** button below to download the necessary libraries, the **code.py** file, and other project files in a zip file.

Connect your board to your computer via a known good data+power USB cable. The board should show up in your File Explorer/Finder (depending on your operating system) as a flash drive named **CIRCUITPY**.

Extract the contents of the zip file, copy the **lib** directory files to **CIRCUITPY/lib**. Copy the **code.py** file to your **CIRCUITPY** drive. The program should self start.

```
1 # SPDX-FileCopyrightText: 2026 Tim Cocks for Adafruit Industries
2 #
```

```

3 # SPDX-License-Identifier: MIT
4 import hashlib
5 import struct
6 import time
7
8 import board
9 import busio
10 import bitmaptools
11 import aesio
12 from displayio import Group, TileGrid, Palette, Bitmap
13 import supervisor
14 import terminalio
15 from adafruit_display_text.text_box import TextBox
16 from adafruit_display_text.bitmap_label import Label
17 import adafruit_binascii
18 import adafruit_gps
19 from digitalio import DigitalInOut, Direction, Pull
20
21 # =====
22 # USER CONFIGURATION
23 # =====
24
25 # --- Sequence of challenges ---
26 # Each entry is a dict with the following keys:
27 #
28 # type : "text" -> Vigenère-encrypted ciphertext
29 #      "image" -> AES-CTR-encrypted .abmp image file
30 #
31 # data : (text) the ciphertext string
32 #      (image) filename of the .abmp.enc file
33 #
34 # sha256 : hex-encoded SHA-256 digest used to confirm correct decryption
35 #      - text -> SHA-256 of the plaintext string encoded as UTF-8
36 #      - image -> SHA-256 of the raw decrypted pixel-data bytes
37 #      Generate these offline with the helper snippet at the bottom
38 #      of this file.
39 #
40 # reading : which sensor data type drives this challenge. One of:
41 #      "gps" -> gps.latitude,gps.longitude (coordinates, string)
42 #
43 # precision_level : Controls how big the "unlock" target area is.
44 #      Approximate sizes at the equator:
45 #      2 -> ~1.1 km (city block scale)
46 #      3 -> ~110 m (large building)
47 #      4 -> ~11 m (room scale)
48 #      Must match the precision used at encryption time.
49 #
50 # iv : (image only) 16-byte AES initialisation vector matching
51 #      the one used during encryption. Omit or set None for text.
52 #
53 # Challenges must be worked through in order: solve #0 to unlock #1, etc.
54 # The key for each challenge is derived from the GPS coordinates at the
55 # target location. The user must physically be there to decrypt correctly.
56
57 SEQUENCE = [
58     # Add more entries here...

```

```

59 ]
60
61 # --- Display rotation (degrees) ---
62 DISPLAY_ROTATION = 180
63
64 # =====
65 # END OF USER CONFIGURATION — do not edit below unless you know what you're doing
66 # =====
67
68 if not SEQUENCE:
69     raise ValueError("SEQUENCE must contain at least one entry.")
70
71 for _i, _entry in enumerate(SEQUENCE):
72     if _entry.get("type") not in ("text", "image"):
73         raise ValueError(f"SEQUENCE[_i]: 'type' must be 'text' or 'image'.")
74     if not _entry.get("data"):
75         raise ValueError(f"SEQUENCE[_i]: 'data' must be set.")
76     if not _entry.get("sha256"):
77         raise ValueError(f"SEQUENCE[_i]: 'sha256' must be set.")
78     if _entry["type"] == "image" and not _entry.get("iv"):
79         raise ValueError(f"SEQUENCE[_i]: image entries require an 'iv'.")
80
81 # Printable ASCII constants (text Vigenère)
82 ASCII_MIN = 32
83 ASCII_MAX = 126
84 ASCII_RANGE = ASCII_MAX - ASCII_MIN + 1 # 95
85
86 # =====
87 # BUTTON SETUP
88 # =====
89
90 btn = DigitalInOut(board.BOOT0)
91 btn.direction = Direction.INPUT
92 btn.pull = Pull.UP
93
94 # btn.value is True when not pressed (pull-up), False when pressed (active-low)
95 btn_prev_value = btn.value
96 btn_last_change_time = time.monotonic()
97
98 # =====
99 # GPS SETUP
100 # =====
101
102 uart = busio.UART(board.TX, board.RX, baudrate=9600, timeout=10)
103 gps = adafruit_gps.GPS(uart, debug=False)
104 gps.send_command(b"PMTK314,0,1,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0")
105 gps.send_command(b"PMTK220,1000")
106
107 # =====
108 # DISPLAY SETUP
109 # =====
110
111 display = supervisor.runtime.display
112 display.rotation = DISPLAY_ROTATION
113
114 main_group = Group()

```

```

115 display.root_group = main_group
116
117 # --- Persistent HUD: progress label (always visible right side) ---
118 hud_group = Group(scale=2, x=2, y=2)
119
120 cur_reading_label = Label(terminalio.FONT)
121 cur_reading_label.anchor_point = (1.0, 1.0)
122 cur_reading_label.anchored_position = (display.width // 2, display.height // 2)
123
124 progress_label = Label(terminalio.FONT)
125 progress_label.anchor_point = (1.0, 1.0)
126 progress_label.anchored_position = (display.width // 2, display.height // 2 - 12)
127
128 hud_group.append(progress_label)
129 hud_group.append(cur_reading_label)
130 main_group.append(hud_group)
131
132
133 # =====
134 # KEY DERIVATION
135 # =====
136
137 def derive_key(coord_str, b64=True):
138     """SHA-256 hash of the GPS coordinate string.
139
140     b64=True -> base64-encoded bytes (Vigenère key)
141     b64=False -> raw 32-byte digest (AES-256 key)
142     """
143     h = hashlib.new("sha256")
144     h.update(coord_str.encode("utf-8"))
145     if b64:
146         return adafruit_binascii.b2a_base64(h.digest()).strip()
147     else:
148         return h.digest()
149
150
151 def get_coord_string():
152     """Return "<lat>,<lon>" at COORD_PRECISION decimal places, or None if no fix."""
153     if not gps.has_fix or gps.latitude is None or gps.longitude is None:
154         return None
155
156     precision_level = entry["precision_level"]
157     _coord_str = f"{gps.latitude:.{precision_level}f},{gps.longitude:.{precision_level}f}"
158     cur_reading_label.text = _coord_str
159     return _coord_str
160
161
162 # =====
163 # HASH VERIFICATION
164 # =====
165
166 def sha256_hex(data):
167     """Return lowercase hex SHA-256 digest of *data* (bytes or str -> UTF-8)."""
168     h = hashlib.new("sha256")
169     if isinstance(data, str):
170         h.update(data.encode("utf-8"))

```

```

171     else:
172         h.update(data)
173     return "".join("{:02x}".format(b) for b in h.digest())
174
175
176 def verify(data, expected_hex):
177     """Return True if SHA-256 of *data* matches *expected_hex*."""
178     return sha256_hex(data) == expected_hex.lower()
179
180
181 # =====
182 # DECRYPTION
183 # =====
184
185 def vigenere_decrypt(ciphertext_str, key_bytes):
186     """Vigenere decryption over printable ASCII (32-126).
187
188     Characters outside that range pass through unchanged.
189     """
190     out = []
191     key_len = len(key_bytes)
192     key_idx = 0
193     for ch in ciphertext_str:
194         c = ord(ch)
195         if ASCII_MIN <= c <= ASCII_MAX:
196             k = key_bytes[key_idx % key_len]
197             k_shifted = (k - ASCII_MIN) % ASCII_RANGE
198             p = ((c - ASCII_MIN) - k_shifted) % ASCII_RANGE
199             out.append(chr(p + ASCII_MIN))
200             key_idx += 1
201         else:
202             out.append(ch)
203     return "".join(out)
204
205
206 def try_decrypt_text(_entry, coord_str):
207     """Decrypt text and verify against the stored hash.
208
209     Returns (success: bool, plaintext: str).
210     Plaintext is always returned so the display shows the garbled attempt
211     while the user is at the wrong location — part of the puzzle UX.
212     """
213     key_bytes = derive_key(coord_str, b64=True)
214     _plaintext = vigenere_decrypt(_entry["data"], key_bytes)
215     _success = verify(_plaintext, _entry["sha256"])
216     return _success, _plaintext
217
218
219 def try_decrypt_image(_entry, coord_str, img_width, img_height,
220                     encrypted_raw, pixel_start):
221     """AES-CTR decrypt pixel data and verify against the stored hash.
222
223     Returns (success: bool, decrypted: bytearray).
224     Decrypted is always returned so the bitmap updates on every coordinate
225     change, letting the user see the image snap into focus at the right spot.
226     """

```

```

227     key_bytes = derive_key(coord_str, b64=False)
228     pixel_data = encrypted_raw[pixel_start: pixel_start + img_width * img_height]
229     _decrypted = bytearray(len(pixel_data))
230     cipher = aesio.AES(key_bytes, aesio.MODE_CTR, IV=_entry["iv"])
231     cipher.decrypt_into(pixel_data, _decrypted)
232     _success = verify(_decrypted, _entry["sha256"])
233     return _success, _decrypted
234
235
236 # =====
237 # CHALLENGE RENDERER
238 # =====
239
240 _image_cache = {}
241
242
243 def load_image_entry(_entry):
244     """Read and parse an encrypted ABMP file. Returns a state dict.
245
246     The file is cached in memory so subsequent challenges that reuse the
247     same filename don't hit the filesystem again.
248     """
249     filename = _entry["data"]
250     if filename not in _image_cache:
251         with open(filename, "rb") as f:
252             raw = bytearray(f.read())
253             _image_cache[filename] = raw
254
255     raw = _image_cache[filename]
256     if raw[0:4] != b"ABMP":
257         raise ValueError(f"Not a valid ABMP file: {filename}")
258
259     img_width, img_height, n_colors = struct.unpack_from("<HHH", raw, 4)
260     palette_start = 10
261     pixel_start = palette_start + n_colors * 3
262
263     palette = Palette(n_colors)
264     for i in range(n_colors):
265         off = palette_start + i * 3
266         r, g, b = raw[off], raw[off + 1], raw[off + 2]
267         palette[i] = (r << 16) | (g << 8) | b
268
269     bitmap = Bitmap(img_width, img_height, n_colors)
270     return {
271         "bitmap": bitmap,
272         "palette": palette,
273         "img_width": img_width,
274         "img_height": img_height,
275         "encrypted_raw": raw,
276         "pixel_start": pixel_start,
277     }
278
279
280 class ChallengeRenderer(Group):
281     """Manages the content_group display for the currently active challenge."""
282

```

```

283 def __init__(self):
284     super().__init__()
285     self._active_index = None
286     self.text_widget = None
287     self.image_state = None
288
289     self.secret_message_text = TextBox(
290         terminalio.FONT,
291         display.width // 2 - 2,
292         (display.height) // 2,
293         align=TextBox.ALIGN_LEFT,
294         scale=2
295     )
296     self.secret_message_text.anchor_point = (0, 0)
297     self.secret_message_text.anchored_position = (2, 0)
298     self.append(self.secret_message_text)
299     self.secret_message_text.text = "Waiting for fix..."
300
301     self.secret_image_tilegrid = None
302
303     self.prompt_text = Label(terminalio.FONT)
304     self.prompt_text.anchor_point = (0, 1.0)
305     self.prompt_text.anchored_position = (2, display.height)
306     self.prompt_text.text = "[ press Boot btn ]"
307     self.prompt_text.hidden = True
308     self.append(self.prompt_text)
309
310 def setup(self, index):
311     """Prepare content_group for challenge *index* (no-op if already set up)."""
312     if self._active_index == index:
313         return
314
315     if self.secret_image_tilegrid is not None and self.secret_image_tilegrid in self:
316         self.remove(self.secret_image_tilegrid)
317
318     self.text_widget = None
319     self.image_state = None
320     _entry = SEQUENCE[index]
321
322     if _entry["type"] == "text":
323         self.secret_message_text.hidden = False
324         self.secret_message_text.text = "Reading sensor..."
325
326     elif _entry["type"] == "image":
327         state = load_image_entry(_entry)
328         self.image_state = state
329         self.secret_message_text.hidden = True
330
331         self.secret_image_tilegrid = TileGrid(state["bitmap"], pixel_shader=state["palette"])
332         # self.secret_image_tilegrid.transpose_xy = True
333         self.append(self.secret_image_tilegrid)
334
335     self._active_index = index
336
337 def update_image(self, decrypted_pixels):
338     """Blit already-decrypted pixel bytes into the bitmap."""

```

```

339     _s = self.image_state
340     bitmaps.tools.arrayblit(
341         _s["bitmap"], decrypted_pixels,
342         x1=0, y1=0, x2=_s["img_width"], y2=_s["img_height"],
343     )
344
345
346     renderer = ChallengeRenderer()
347     main_group.append(renderer)
348
349     # =====
350     # HUD + COMPLETION
351     # =====
352
353     def update_hud(_current_index, _total):
354         """Update the top progress bar label."""
355         label = f"{_current_index + 1}/{_total}"
356         if progress_label.text != label:
357             progress_label.text = label
358
359
360     def show_completion_screen():
361         """Replace content_group with a 'you win' message and halt."""
362         while len(renderer):
363             renderer.pop()
364
365         fin = TextBox(
366             terminalio.FONT,
367             display.width // 2,
368             (display.height - 16) // 2,
369             align=TextBox.ALIGN_CENTER,
370             scale=2
371         )
372         fin.anchor_point = (0, 0)
373         fin.anchored_position = (0, 0)
374         fin.text = "All secrets revealed.\nGood job!"
375         renderer.append(fin)
376         while True:
377             pass # halt / wait forever
378
379
380     # =====
381     # MAIN LOOP
382     # =====
383
384     current_index = 0
385     old_coord = ""
386     total = len(SEQUENCE)
387     challenge_solved = False # True while waiting for the user to press the button
388
389     while True:
390         gps.update()
391
392         entry = SEQUENCE[current_index]
393         renderer.setup(current_index)
394         update_hud(current_index, total)

```

```

395
396 now = time.monotonic()
397 cur_coord = get_coord_string() # None until GPS has a fix
398
399 # -----
400 # While a challenge is solved-but-not-yet-confirmed, keep updating
401 # the display with the latest decryption (GPS may still drift slightly)
402 # but wait for a button press before advancing.
403 # -----
404 if challenge_solved:
405     btn_cur_val = btn.value
406     if not btn_cur_val and btn_prev_value:
407         print(f"Challenge {current_index + 1} confirmed by button press — advancing.")
408         current_index += 1
409         challenge_solved = False
410         old_coord = "" # force re-decrypt immediately on next loop
411
412         if current_index >= total:
413             update_hud(total, total)
414             show_completion_screen()
415         btn_prev_value = btn_cur_val
416         continue # skip normal decrypt logic while waiting for button
417
418 # -----
419 # No GPS fix — show waiting message and loop.
420 # -----
421 if cur_coord is None:
422     if old_coord != "":
423         # Fix was just lost
424         if entry["type"] == "text":
425             renderer.secret_message_text.text = "Waiting for fix..."
426             old_coord = ""
427         continue
428
429 # -----
430 # Normal path: decrypt on every coordinate change and check for a solve.
431 # -----
432 if cur_coord != old_coord:
433     print(f"[{current_index + 1}]{total} coord: {cur_coord}")
434     old_coord = cur_coord
435
436     if entry["type"] == "text":
437         success, plaintext = try_decrypt_text(entry, cur_coord)
438         renderer.secret_message_text.text = plaintext
439
440         if success:
441             print(f"Challenge {current_index + 1} SOLVED (text): {plaintext}")
442             challenge_solved = True
443
444     elif entry["type"] == "image":
445         s = renderer.image_state
446         success, decrypted = try_decrypt_image(
447             entry, cur_coord,
448             s["img_width"], s["img_height"],
449             s["encrypted_raw"], s["pixel_start"],
450         )

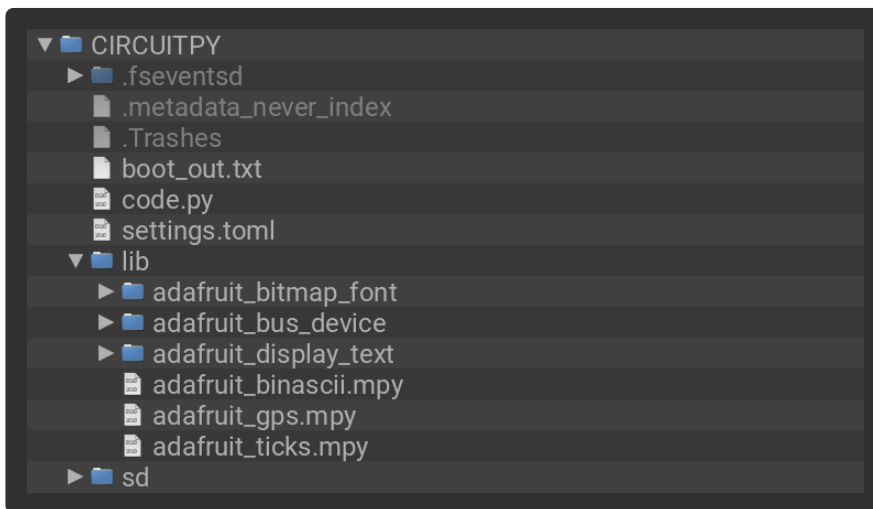
```

```
451     renderer.update_image(decrypted)
452
453     if success:
454         print(f"Challenge {current_index + 1} SOLVED (image)")
455         challenge_solved = True
```

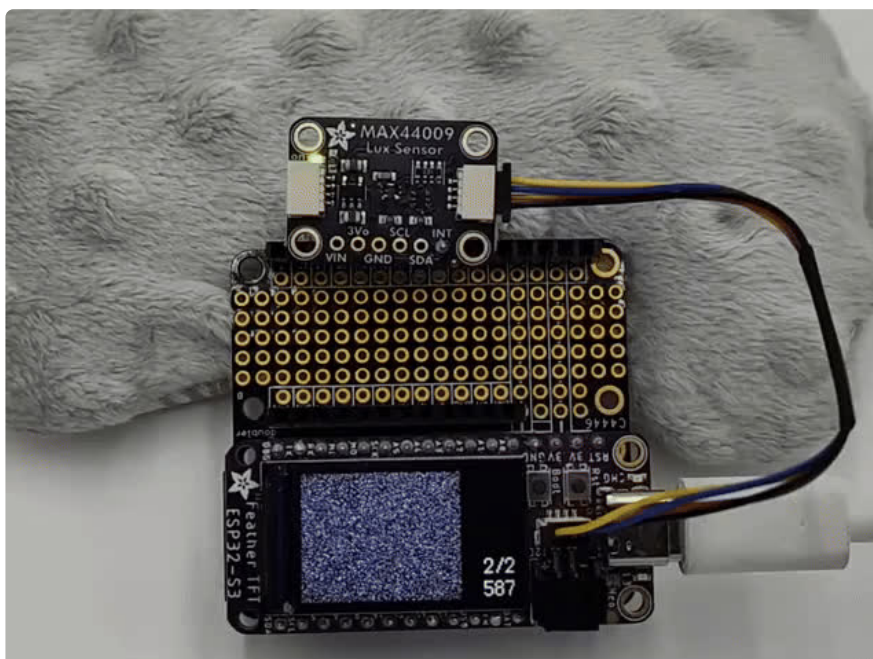
https://github.com/adafruit/Adafruit_Learning_System_Guides/blob/main/Sensor_Locked_Secrets/gps/code.py

Drive Structure

After copying the files, your drive should look like the listing below. It can contain other files as well, but must contain these at a minimum.



Lux Level



Encrypt Secrets

Encrypt your secret messages using the process documented on [this guide page](https://adafru.it/1aDx) (<https://adafru.it/1aDx>). Choose **lux(lux)** as the reading type. A precision level of **100** or **50** works well for lux readings on the MAX44009.

<https://adafru.it/1aDv>

Code

To use the application, you need to obtain **code.py** with the program, and the other project files to place on the Feather **CIRCUITPY** drive.

Thankfully, this can be done in one go. In the example below, click the **Download Project Bundle** button below to download the necessary libraries, the **code.py** file, and other project files in a zip file.

Connect your board to your computer via a known good data+power USB cable. The board should show up in your File Explorer/Finder (depending on your operating system) as a flash drive named **CIRCUITPY**.

Extract the contents of the zip file, copy the **lib** directory files to **CIRCUITPY/lib**. Copy the **code.py** file to your **CIRCUITPY** drive. The program should self start.

```
1 # SPDX-FileCopyrightText: 2026 Tim Cocks for Adafruit Industries
2 #
3 # SPDX-License-Identifier: MIT
4 import hashlib
5 import struct
6 import time
7
8 import board
9 import bitmaptools
10 import aesio
11 from displayio import Group, TileGrid, Palette, Bitmap
12 import supervisor
13 import terminalio
14 from adafruit_display_text.text_box import TextBox
15 from adafruit_display_text.bitmap_label import Label
16 import adafruit_binascii
17 from adafruit_max44009 import MAX44009
18 from digitalio import DigitalInOut, Direction, Pull
19
20 # =====
21 # USER CONFIGURATION
22 # =====
23
24 # --- Sequence of challenges ---
```

```

25 # Each entry is a dict with the following keys:
26 #
27 # type      : "text" -> Vigenère-encrypted ciphertext
28 #          "image" -> AES-CTR-encrypted .abmp image file
29 #
30 # data      : (text) the ciphertext string
31 #          (image) filename of the .abmp.enc file
32 #
33 # sha256    : hex-encoded SHA-256 digest used to confirm correct decryption
34 #          - text -> SHA-256 of the plaintext string encoded as UTF-8
35 #          - image -> SHA-256 of the raw decrypted pixel-data bytes
36 #          Generate these offline with the encryptor web page.
37 #
38 # reading   : which sensor data type drives this challenge. One of:
39 #          "lux" -> sensor.lux (lux, float)
40 #
41 # precision_level : width of each unlock band, in the units of `reading`.
42 #          e.g. reading=lux, precision_level=1 -> bands like 22-23
43 #          reading=lux, precision_level=5 -> bands like 40-45
44 #          reading=lux, precision_level=100 -> bands like 400-500
45 #          Must match the precision used at encryption time.
46 #
47 # iv       : (image only) 16-byte AES initialisation vector matching
48 #          the one used during encryption. "InitializationVe" is used
49 #          by default if None. Omit or set None for text.
50 #
51 # Challenges must be worked through in order: solve #0 to unlock #1, etc.
52 #
53
54
55 SEQUENCE = [
56     {
57         "type":      "text",
58         "data":      "?[4&@izK<!=?6s-'5URJf:Jz^:azh$",
59         "sha256":    "1dbc88f6f952ec922ebe4343b5fdfe15919215fefabbd48863c2f1e910a6a118",
60         "reading":   "lux",
61         "precision_level": 100,
62     },
63     {
64         "type":      "text",
65         "data":      "C:Z)!*t~6L:)ox|=l4*c=;gB:.[wt`<g@",
66         "sha256":    "bd4a361c0ae2039e04f82281a91cc1f05097562e296d211afce72145af71cbab",
67         "reading":   "lux",
68         "precision_level": 100,
69     },
70     # Add more entries here...
71 ]
72
73 # Map of reading-type names to (sensor attribute, unit label for prints).
74 READING_TYPES = {
75     "lux": ("lux", "lux"),
76 }
77
78 # --- Display rotation (degrees) ---
79 DISPLAY_ROTATION = 180
80

```

```

81 # =====
82 # END OF USER CONFIGURATION
83 # =====
84
85 # =====
86 # SENSOR SETUP
87 # =====
88
89 i2c = board.I2C()
90 sensor = MAX44009(i2c)
91 SENSOR_READ_COOLDOWN = 0.25
92
93 # =====
94 # Validate SEQUENCE entries
95 # =====
96 if not SEQUENCE:
97     raise ValueError("SEQUENCE must contain at least one entry.")
98
99 for _i, _entry in enumerate(SEQUENCE):
100     if _entry.get("type") not in ("text", "image"):
101         raise ValueError(f"SEQUENCE[_i]: 'type' must be 'text' or 'image'.")
102     if not _entry.get("data"):
103         raise ValueError(f"SEQUENCE[_i]: 'data' must be set.")
104     if not _entry.get("sha256"):
105         raise ValueError(f"SEQUENCE[_i]: 'sha256' must be set.")
106     if _entry.get("reading") not in READING_TYPES:
107         raise ValueError(
108             f"SEQUENCE[_i]: 'reading' must be one of {list(READING_TYPES)}."
109         )
110     _bs = _entry.get("precision_level")
111     if not isinstance(_bs, int) or _bs <= 0:
112         raise ValueError(f"SEQUENCE[_i]: 'precision_level' must be a positive int.")
113     if _entry["type"] == "image" and not _entry.get("iv"):
114         SEQUENCE[_i]["iv"] = b"InitializationVe"
115
116 # Printable ASCII constants (text Vigenere)
117 ASCII_MIN = 32
118 ASCII_MAX = 126
119 ASCII_RANGE = ASCII_MAX - ASCII_MIN + 1 # 95
120
121 # =====
122 # BUTTON SETUP
123 # =====
124
125 btn = DigitalInOut(board.BOOT0)
126 btn.direction = Direction.INPUT
127 btn.pull = Pull.UP
128
129 # btn.value is True when not pressed (pull-up), False when pressed (active-low)
130 btn_prev_value = btn.value # tracks last reading for edge detection
131
132 # =====
133 # DISPLAY SETUP
134 # =====
135
136 display = supervisor.runtime.display

```

```

137 display.rotation = DISPLAY_ROTATION
138
139 main_group = Group()
140 display.root_group = main_group
141
142 # --- Persistent HUD: progress label (always visible right side) ---
143 hud_group = Group(scale=2, x=2, y=2)
144
145 cur_reading_label = Label(terminalio.FONT)
146 cur_reading_label.anchor_point = (1.0, 1.0)
147 cur_reading_label.anchored_position = (display.width // 2, display.height // 2)
148
149 progress_label = Label(terminalio.FONT)
150 progress_label.anchor_point = (1.0, 1.0)
151 progress_label.anchored_position = (display.width // 2, display.height // 2 - 12)
152
153 hud_group.append(progress_label)
154 hud_group.append(cur_reading_label)
155 main_group.append(hud_group)
156
157
158 # =====
159 # KEY DERIVATION
160 # =====
161
162 def derive_key(range_str, b64=True):
163     """SHA-256 of the sensor reading range string.
164
165     b64=True -> base64-encoded bytes (Vigenere key)
166     b64=False -> raw 32-byte digest (AES-256 key)
167     """
168     h = hashlib.new("sha256")
169     h.update(range_str.encode("utf-8"))
170     if b64:
171         return adafruit_binascii.b2a_base64(h.digest()).strip()
172     else:
173         return h.digest()
174
175
176 def get_range_string_from_sensor(_entry):
177     """Return a bucketed range string like '22-23' for the given challenge entry.
178
179     Reads whichever sensor attribute is named by entry['reading'] and buckets
180     the (truncated-to-int) value using entry['precision_level'].
181     """
182     attr_name, _unit = READING_TYPES[_entry["reading"]]
183
184     reading = int(getattr(sensor, attr_name))
185     cur_reading_label.text = str(reading)
186     precision_level = _entry["precision_level"]
187     bucket = (reading // precision_level) * precision_level
188     return f"[bucket]-[bucket + precision_level]"
189
190
191 # =====
192 # HASH VERIFICATION

```

```

193 # =====
194
195 def sha256_hex(data):
196     """Return lowercase hex SHA-256 digest of *data* (bytes or str -> UTF-8)."""
197     h = hashlib.new("sha256")
198     if isinstance(data, str):
199         h.update(data.encode("utf-8"))
200     else:
201         h.update(data)
202     # Convert raw digest bytes to hex without binascii
203     return "".join("{:02x}".format(b) for b in h.digest())
204
205
206 def verify(data, expected_hex):
207     """Return True if SHA-256 of *data* matches *expected_hex*."""
208     return sha256_hex(data) == expected_hex.lower()
209
210
211 # =====
212 # DECRYPTION
213 # =====
214
215 def vigenere_decrypt(ciphertext_str, key_bytes):
216     """Vigenere decryption over printable ASCII (32-126).
217
218     Characters outside that range pass through unchanged.
219     """
220     out = []
221     key_len = len(key_bytes)
222     key_idx = 0
223     for ch in ciphertext_str:
224         c = ord(ch)
225         if ASCII_MIN <= c <= ASCII_MAX:
226             k = key_bytes[key_idx % key_len]
227             k_shifted = (k - ASCII_MIN) % ASCII_RANGE
228             p = ((c - ASCII_MIN) - k_shifted) % ASCII_RANGE
229             out.append(chr(p + ASCII_MIN))
230             key_idx += 1
231         else:
232             out.append(ch)
233     return "".join(out)
234
235
236 def try_decrypt_text(_entry, range_str):
237     """Decrypt text and verify against the stored hash.
238
239     Returns (success: bool, plaintext: str).
240     plaintext is always returned so the display shows the attempt in progress
241     (garbled text while the wrong sensor reading is active is part of the puzzle UX).
242     """
243     key_bytes = derive_key(range_str, b64=True)
244     _plaintext = vigenere_decrypt(_entry["data"], key_bytes)
245     _success = verify(_plaintext, _entry["sha256"])
246     return _success, _plaintext
247
248

```

```

249 def try_decrypt_image(_entry, range_str, img_width, img_height,
250     encrypted_raw, pixel_start):
251     """AES-CTR decrypt pixel data and verify against the stored hash.
252
253     Returns (success: bool, decrypted: bytearray).
254     decrypted is always returned so the bitmap updates on every reading change,
255     letting the user see the image snap into focus at the correct sensor level.
256     """
257     key_bytes = derive_key(range_str, b64=False)
258     pixel_data = encrypted_raw[pixel_start: pixel_start + img_width * img_height]
259     _decrypted = bytearray(len(pixel_data))
260     cipher = aesio.AES(key_bytes, aesio.MODE_CTR, IV=_entry["iv"])
261     cipher.decrypt_into(pixel_data, _decrypted)
262     _success = verify(_decrypted, _entry["sha256"])
263     return _success, _decrypted
264
265
266 # =====
267 # CHALLENGE RENDERER
268 # =====
269
270 _image_cache = {}
271
272
273 def load_image_entry(_entry):
274     """Read and parse an encrypted ABMP file. Returns a state dict."""
275     filename = _entry["data"]
276     if filename not in _image_cache:
277         with open(filename, "rb") as f:
278             raw = bytearray(f.read())
279             _image_cache[filename] = raw
280
281     raw = _image_cache[filename]
282     if raw[0:4] != b"ABMP":
283         raise ValueError(f"Not a valid ABMP file: {filename}")
284
285     img_width, img_height, n_colors = struct.unpack_from("<HHH", raw, 4)
286     palette_start = 10
287     pixel_start = palette_start + n_colors * 3
288
289     palette = Palette(n_colors)
290     for i in range(n_colors):
291         off = palette_start + i * 3
292         r, g, b = raw[off], raw[off + 1], raw[off + 2]
293         palette[i] = (r << 16) | (g << 8) | b
294
295     bitmap = Bitmap(img_width, img_height, n_colors)
296     return {
297         "bitmap": bitmap,
298         "palette": palette,
299         "img_width": img_width,
300         "img_height": img_height,
301         "encrypted_raw": raw,
302         "pixel_start": pixel_start,
303     }
304

```

```

305
306 class ChallengeRenderer(Group):
307     """Manages the content_group display for the currently active challenge."""
308
309     def __init__(self):
310         super().__init__()
311         self._active_index = None
312         self.text_widget = None
313         self.image_state = None
314
315         self.secret_message_text = TextBox(
316             terminalio.FONT,
317             display.width // 2 - 2,
318             (display.height) // 2,
319             align=TextBox.ALIGN_LEFT,
320             scale=2
321         )
322         self.secret_message_text.anchor_point = (0, 0)
323         self.secret_message_text.anchored_position = (2, 0)
324         self.append(self.secret_message_text)
325         self.secret_message_text.text = "Reading sensor..."
326
327         self.secret_image_tilegrid = None
328
329         self.prompt_text = Label(terminalio.FONT)
330         self.prompt_text.anchor_point = (0, 1.0)
331         self.prompt_text.anchored_position = (2, display.height)
332         self.prompt_text.text = "[ press Boot btn ]"
333         self.prompt_text.hidden = True
334         self.append(self.prompt_text)
335
336     def setup(self, index):
337         """Prepare content_group for challenge *index* (no-op if already set up)."""
338         if self._active_index == index:
339             return
340
341         if self.secret_image_tilegrid is not None and self.secret_image_tilegrid in self:
342             self.remove(self.secret_image_tilegrid)
343
344         self.text_widget = None
345         self.image_state = None
346         _entry = SEQUENCE[index]
347
348         if _entry["type"] == "text":
349             self.secret_message_text.hidden = False
350             self.secret_message_text.text = "Reading sensor..."
351
352         elif _entry["type"] == "image":
353             state = load_image_entry(_entry)
354             self.image_state = state
355             self.secret_message_text.hidden = True
356
357             self.secret_image_tilegrid = TileGrid(state["bitmap"], pixel_shader=state["palette"])
358             # self.secret_image_tilegrid.transpose_xy = True
359             self.append(self.secret_image_tilegrid)
360

```

```

361     self._active_index = index
362
363     def update_image(self, decrypted_pixels):
364         """Blit already-decrypted pixel bytes into the bitmap."""
365         _s = self.image_state
366         bitmaptools.arrayblit(
367             _s["bitmap"], decrypted_pixels,
368             x1=0, y1=0, x2=_s["img_width"], y2=_s["img_height"],
369         )
370
371
372     renderer = ChallengeRenderer()
373     main_group.append(renderer)
374
375
376     # =====
377     # HUD + COMPLETION
378     # =====
379
380     def update_hud(_current_index, _total):
381         """Update the top progress bar label."""
382         label = f"[_current_index + 1]/[_total]"
383         if progress_label.text != label:
384             progress_label.text = label
385
386
387     def show_completion_screen():
388         """Replace challenge content with a 'you win' message and halt."""
389         while len(renderer):
390             renderer.pop()
391
392         fin = TextBox(
393             terminalio.FONT,
394             display.width // 2,
395             (display.height - 16) // 2,
396             align=TextBox.ALIGN_CENTER,
397             scale=2
398         )
399         fin.anchor_point = (0, 0)
400         fin.anchored_position = (0, 0)
401         fin.text = "All secrets revealed.\nGood job!"
402
403         renderer.append(fin)
404         hud_group.hidden = True
405         while True:
406             pass # halt / wait forever
407
408
409     # =====
410     # MAIN LOOP
411     # =====
412
413     current_index = 0
414     old_range = ""
415     total = len(SEQUENCE)
416     challenge_solved = False # True while waiting for the user to press the button

```

```

417
418 last_sensor_read_time = 0
419 cur_cache_key = None
420 while True:
421     entry = SEQUENCE[current_index]
422     now = time.monotonic()
423     if now - last_sensor_read_time > SENSOR_READ_COOLDOWN and not challenge_solved:
424         try:
425             cur_range = get_range_string_from_sensor(entry)
426         except OSError as e:
427             print(e, "retrying after cooldown")
428             cur_range = cur_cache_key.split(":-1")
429             last_sensor_read_time = now
430     else:
431         cur_range = cur_cache_key.split(":-1")
432     # Include reading type in cache key so changing challenge boundaries
433     # don't accidentally match a previous challenge's bucket value.
434     cur_cache_key = f"{entry['reading']}:{cur_range}"
435
436     renderer.setup(current_index)
437     update_hud(current_index, total)
438
439     # -----
440     # While a challenge is solved-but-not-yet-confirmed, keep updating
441     # the display with the latest decryption (sensor reading may still drift) but
442     # wait for a button press before advancing.
443     # -----
444     if challenge_solved:
445         btn_cur_val = btn.value
446         if not btn_cur_val and btn_prev_value:
447             print(f"Challenge {current_index + 1} confirmed by button press — advancing.")
448             current_index += 1
449             challenge_solved = False
450             old_range = "" # force re-decrypt immediately on next loop
451
452         if current_index >= total:
453             update_hud(total, total)
454             show_completion_screen()
455             btn_prev_value = btn_cur_val
456             continue # skip normal decrypt logic while waiting for button
457
458     # -----
459     # Normal path: decrypt on every reading-range change and check for a solve.
460     # -----
461     now = time.monotonic()
462     if cur_cache_key != old_range:
463         last_sensor_read_time = now
464         unit = READING_TYPES[entry["reading"]][1]
465         print(f"[{current_index + 1}/{total}] {entry['reading']} range: {cur_range} {unit}")
466         old_range = cur_cache_key
467
468         if entry["type"] == "text":
469             success, plaintext = try_decrypt_text(entry, cur_range)
470             renderer.secret_message_text.text = plaintext
471
472         if success:

```

```

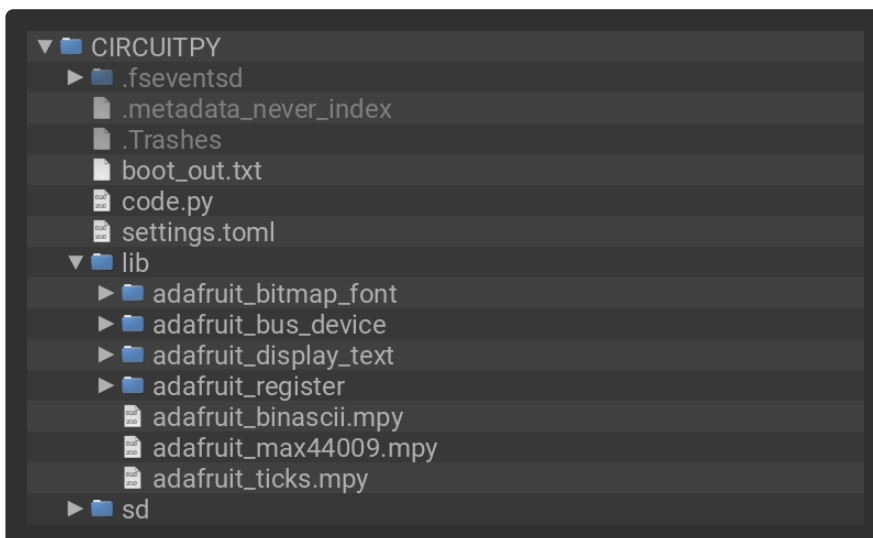
473     print(f"Challenge {current_index + 1} SOLVED (text): {plaintext}")
474     cur_reading_label.text = "*" + cur_reading_label.text
475     challenge_solved = True
476
477     elif entry["type"] == "image":
478         s = renderer.image_state
479         success, decrypted = try_decrypt_image(
480             entry, cur_range,
481             s["img_width"], s["img_height"],
482             s["encrypted_raw"], s["pixel_start"],
483         )
484         renderer.update_image(decrypted)
485
486         if success:
487             print(f"Challenge {current_index + 1} SOLVED (image)")
488             cur_reading_label.text = "*" + cur_reading_label.text
489             challenge_solved = True

```

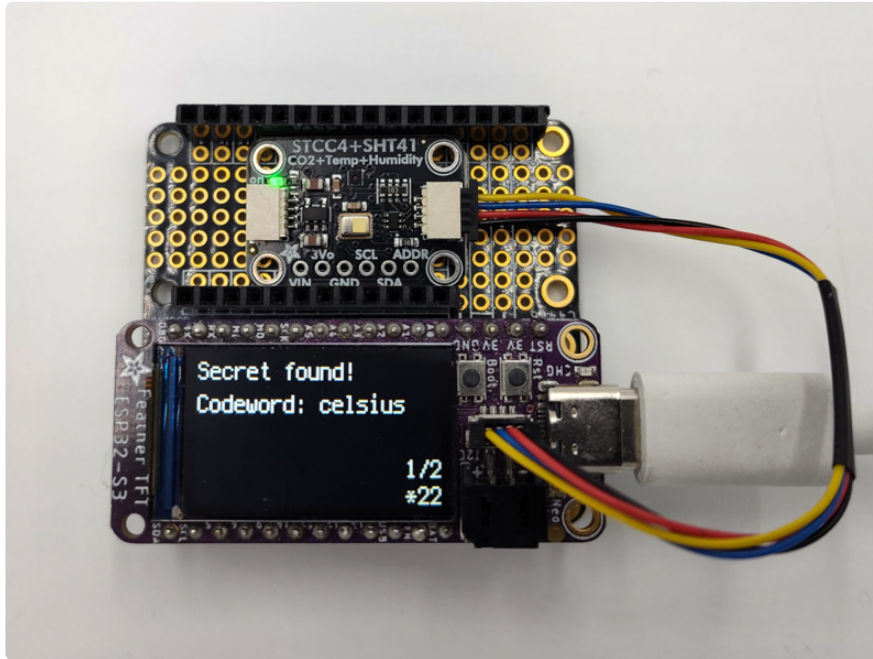
https://github.com/adafruit/Adafruit_Learning_System_Guides/blob/main/Sensor_Locked_Secrets/max44009/code.py

Drive Structure

After copying the files, your drive should look like the listing below. It can contain other files as well, but must contain these at a minimum.



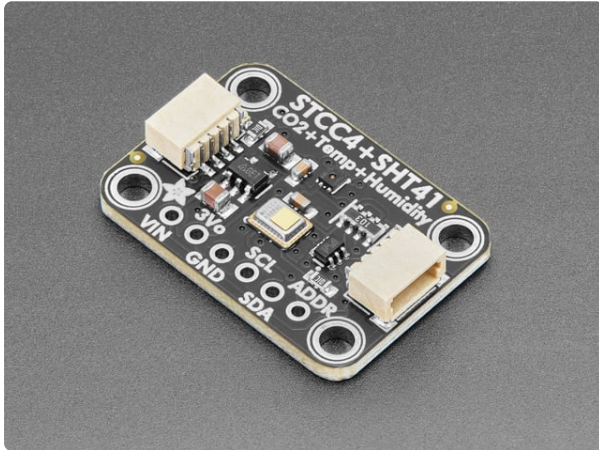
Temperature Humidity CO2



This version of the code uses the STCC4+SHT41 breakout to decrypt secrets with temperature, humidity, or CO2 readings. Since all 3 types of data come from the same sensor, it is possible to use a sequence of secrets that are each encrypted with different reading types.

One example usage for this is setting up a scavenger hunt-like experience where participants are encouraged to observe changes to the visible sensor reading and experiment with different ways to influence the sensor or environment to find each secret. The revealed text or image can contain a clue that leads them towards the solution for the next puzzle in the sequence.

Hardware



Adafruit STCC4 and SHT41 - CO2, Temperature & Humidity Sensor

CO₂ sensors are essential for determining if a room is too 'stuffy' - high CO₂ makes humans grumpy and tired. That's why it's always nice to take a deep breath...

<https://www.adafruit.com/product/6478>

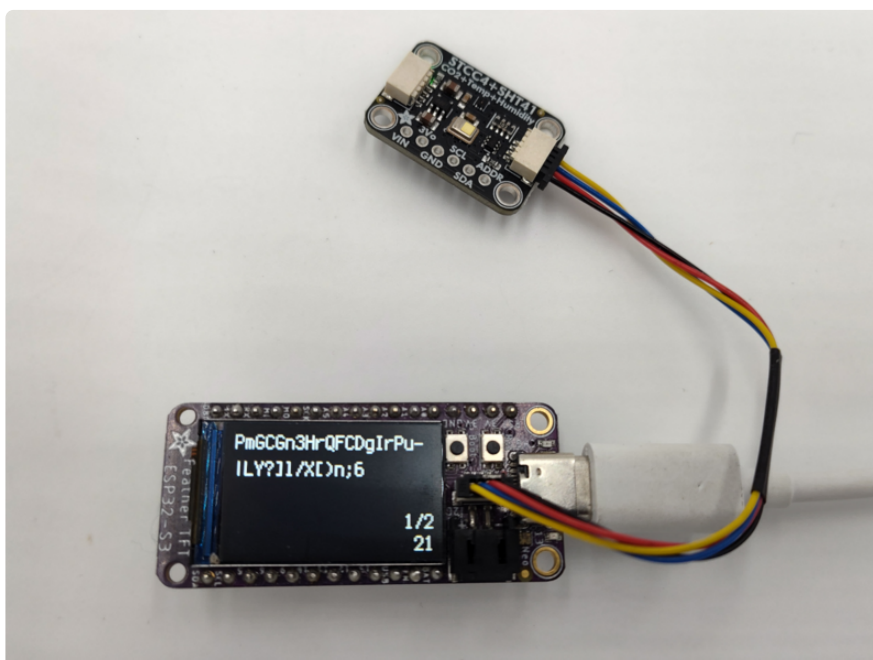


STEMMA QT / Qwiic JST SH 4-pin Cable - 100mm Long

This 4-wire cable is a little over 100mm / 4" long and fitted with JST-SH female 4-pin connectors on both ends. Compared with the chunkier JST-PH these are 1mm pitch instead of...

<https://www.adafruit.com/product/4210>

Connect the STCC4+SHT41 breakout to the Feather S3 TFT with a STEMMA QT cable.



Encrypt Secrets

Encrypt your secret messages using the process documented on [this guide page](https://adafru.it/1aDx) (<https://adafru.it/1aDx>). Choose **temperature**, **humidity**, or **CO2** as desired for each secret that you encrypt. Temperature and humidity readings use Celsius and percentage units respectively. Both work well with a precision level of **1**, but you can use a larger value if you want make the target range easier to find. CO2 uses PPM which is likely to have a value in the range of 300-1000 for normal environments, so a larger precision level of **100** or **50** works better for it.



It is unhealthy for humans to spend time in environments with CO2 level over 1000ppm. A base line reading in clean outdoor air is typically around 400. CO2 based encryption should use a target level of 1000 or less so that it does not encourage participants to create or venture into unsafe environments.

<https://adafru.it/1aDv>

Code

To use the application, you need to obtain **code.py** with the program, and the other project files to place on the Feather **CIRCUITPY** drive.

Thankfully, this can be done in one go. In the example below, click the **Download Project Bundle** button below to download the necessary libraries, the **code.py** file, and other project files in a zip file.

Connect your board to your computer via a known good data+power USB cable. The board should show up in your File Explorer/Finder (depending on your operating system) as a flash drive named **CIRCUITPY**.

Extract the contents of the zip file, copy the **lib** directory files to **CIRCUITPY/lib**. Copy the **code.py** file to your **CIRCUITPY** drive. The program should self start.

```
1 # SPDX-FileCopyrightText: 2026 Tim Cocks for Adafruit Industries
2 #
3 # SPDX-License-Identifier: MIT
4 import hashlib
5 import struct
6 import time
7
```

```

8 import board
9 import bitmaptools
10 import aesio
11 from displayio import Group, TileGrid, Palette, Bitmap
12 import supervisor
13 import terminalio
14 from adafruit_display_text.text_box import TextBox
15 from adafruit_display_text.bitmap_label import Label
16 import adafruit_binascii
17 import adafruit_stcc4
18 from digitalio import DigitalInOut, Direction, Pull
19
20 # =====
21 # USER CONFIGURATION
22 # =====
23
24 # --- Sequence of challenges ---
25 # Each entry is a dict with the following keys:
26 #
27 # type      : "text" -> Vigenère-encrypted ciphertext
28 #           "image" -> AES-CTR-encrypted .abmp image file
29 #
30 # data      : (text) the ciphertext string
31 #           (image) filename of the .abmp.enc file
32 #
33 # sha256    : hex-encoded SHA-256 digest used to confirm correct decryption
34 #           - text -> SHA-256 of the plaintext string encoded as UTF-8
35 #           - image -> SHA-256 of the raw decrypted pixel-data bytes
36 #           Generate these offline with the encryptor web page.
37 #
38 # reading   : which sensor data type drives this challenge. One of:
39 #           "temperature" -> sensor.temperature (°C, float)
40 #           "humidity" -> sensor.relative_humidity (% , float)
41 #           "co2" -> sensor.CO2 (ppm, int)
42 #
43 # precision_level : width of each unlock band, in the units of `reading`.
44 #           e.g. reading=temperature, precision_level=1 -> bands like 22-23 (°C)
45 #           reading=humidity, precision_level=5 -> bands like 40-45 (%)
46 #           reading=co2, precision_level=100 -> bands like 400-500 (ppm)
47 #           Must match the precision used at encryption time.
48 #
49 # iv       : (image only) 16-byte AES initialisation vector matching
50 #           the one used during encryption. "InitializationVe" is used
51 #           by default if None. Omit or set None for text.
52 #
53 # Challenges must be worked through in order: solve #0 to unlock #1, etc.
54 #
55
56
57 SEQUENCE = [
58     {
59         "type": "text",
60         "data": ">H5mv=hlbL3&iW4Fu]gbBNQ5&.2YC`J",
61         "sha256": "9313a63935c54a01de05e00bca67176639101f6251e411a7422ea1c0451a2588",
62         "reading": "temperature",
63         "precision_level": 2,

```

```

64     },
65     {
66         "type":    "image",
67         "data":    "example_image.abmp.enc",
68         "iv":      b"InitializationVe",
69         "sha256":  "932185ffbb8b245a97a0819428d6038997e968174cadd3bd0a5bab970d0e560",
70         "reading": "humidity",
71         "precision_level": 1,
72     },
73     # Add more entries here...
74 ]
75
76 # Map of reading-type names to (sensor attribute, unit label for prints).
77 READING_TYPES = {
78     "temperature": ("temperature", "°C"),
79     "humidity": ("relative_humidity", "%"),
80     "co2": ("CO2", "ppm"),
81 }
82
83 # --- Display rotation (degrees) ---
84 DISPLAY_ROTATION = 180
85
86 # =====
87 # END OF USER CONFIGURATION
88 # =====
89
90 # =====
91 # SENSOR SETUP
92 # =====
93
94 i2c = board.I2C()
95 sensor = adafruit_stcc4.STCC4(i2c)
96 sensor.continuous_measurement = True
97 SENSOR_READ_COOLDOWN = 1.0
98
99 # =====
100 # Validate SEQUENCE entries
101 # =====
102 if not SEQUENCE:
103     raise ValueError("SEQUENCE must contain at least one entry.")
104
105 for _i, _entry in enumerate(SEQUENCE):
106     if _entry.get("type") not in ("text", "image"):
107         raise ValueError(f"SEQUENCE[{_i}]: 'type' must be 'text' or 'image'.")
108     if not _entry.get("data"):
109         raise ValueError(f"SEQUENCE[{_i}]: 'data' must be set.")
110     if not _entry.get("sha256"):
111         raise ValueError(f"SEQUENCE[{_i}]: 'sha256' must be set.")
112     if _entry.get("reading") not in READING_TYPES:
113         raise ValueError(
114             f"SEQUENCE[{_i}]: 'reading' must be one of {list(READING_TYPES)}."
115         )
116     _bs = _entry.get("precision_level")
117     if not isinstance(_bs, int) or _bs <= 0:
118         raise ValueError(f"SEQUENCE[{_i}]: 'precision_level' must be a positive int.")
119     if _entry["type"] == "image" and not _entry.get("iv"):

```

```

120     SEQUENCE[_i][iv] = b"InitializationVe"
121
122     # Printable ASCII constants (text Vigenere)
123     ASCII_MIN = 32
124     ASCII_MAX = 126
125     ASCII_RANGE = ASCII_MAX - ASCII_MIN + 1 # 95
126
127     # =====
128     # BUTTON SETUP
129     # =====
130
131     btn = DigitalInOut(board.BOOT0)
132     btn.direction = Direction.INPUT
133     btn.pull = Pull.UP
134
135     # btn.value is True when not pressed (pull-up), False when pressed (active-low)
136     btn_prev_value = btn.value # tracks last reading for edge detection
137
138     # =====
139     # DISPLAY SETUP
140     # =====
141
142     display = supervisor.runtime.display
143     display.rotation = DISPLAY_ROTATION
144
145     main_group = Group()
146     display.root_group = main_group
147
148     # --- Persistent HUD: progress label (always visible right side) ---
149     hud_group = Group(scale=2, x=2, y=2)
150
151     cur_reading_label = Label(terminalio.FONT)
152     cur_reading_label.anchor_point = (1.0, 1.0)
153     cur_reading_label.anchored_position = (display.width // 2, display.height // 2)
154
155     progress_label = Label(terminalio.FONT)
156     progress_label.anchor_point = (1.0, 1.0)
157     progress_label.anchored_position = (display.width // 2, display.height // 2 - 12)
158
159     hud_group.append(progress_label)
160     hud_group.append(cur_reading_label)
161     main_group.append(hud_group)
162
163
164     # =====
165     # KEY DERIVATION
166     # =====
167
168     def derive_key(range_str, b64=True):
169         """SHA-256 of the sensor reading range string.
170
171         b64=True -> base64-encoded bytes (Vigenere key)
172         b64=False -> raw 32-byte digest (AES-256 key)
173         """
174         h = hashlib.new("sha256")
175         h.update(range_str.encode("utf-8"))

```

```

176     if b64:
177         return adafruit_binascii.b2a_base64(h.digest()).strip()
178     else:
179         return h.digest()
180
181
182 def get_range_string_from_sensor(_entry):
183     """Return a bucketed range string like '22-23' for the given challenge entry.
184
185     Reads whichever sensor attribute is named by entry['reading'] and buckets
186     the (truncated-to-int) value using entry['precision_level'].
187     """
188     attr_name, _unit = READING_TYPES[_entry["reading"]]
189
190     if attr_name != "CO2":
191         # read CO2 to refresh other data types
192         _ = sensor.CO2
193
194     reading = int(getattr(sensor, attr_name))
195     cur_reading_label.text = str(reading)
196     precision_level = _entry["precision_level"]
197     bucket = (reading // precision_level) * precision_level
198     return f"{bucket}-{bucket + precision_level}"
199
200
201 # =====
202 # HASH VERIFICATION
203 # =====
204
205 def sha256_hex(data):
206     """Return lowercase hex SHA-256 digest of *data* (bytes or str -> UTF-8)."""
207     h = hashlib.new("sha256")
208     if isinstance(data, str):
209         h.update(data.encode("utf-8"))
210     else:
211         h.update(data)
212     # Convert raw digest bytes to hex without binascii
213     return "".join("{:02x}".format(b) for b in h.digest())
214
215
216 def verify(data, expected_hex):
217     """Return True if SHA-256 of *data* matches *expected_hex*."""
218     return sha256_hex(data) == expected_hex.lower()
219
220
221 # =====
222 # DECRYPTION
223 # =====
224
225 def vigenere_decrypt(ciphertext_str, key_bytes):
226     """Vigenere decryption over printable ASCII (32-126).
227
228     Characters outside that range pass through unchanged.
229     """
230     out = []
231     key_len = len(key_bytes)

```

```

232     key_idx = 0
233     for ch in ciphertext_str:
234         c = ord(ch)
235         if ASCII_MIN <= c <= ASCII_MAX:
236             k = key_bytes[key_idx % key_len]
237             k_shifted = (k - ASCII_MIN) % ASCII_RANGE
238             p = ((c - ASCII_MIN) - k_shifted) % ASCII_RANGE
239             out.append(chr(p + ASCII_MIN))
240             key_idx += 1
241         else:
242             out.append(ch)
243     return "".join(out)
244
245
246 def try_decrypt_text(_entry, range_str):
247     """Decrypt text and verify against the stored hash.
248
249     Returns (success: bool, plaintext: str).
250     plaintext is always returned so the display shows the attempt in progress
251     (garbled text while the wrong sensor reading is active is part of the puzzle UX).
252     """
253     key_bytes = derive_key(range_str, b64=True)
254     _plaintext = vigenere_decrypt(_entry["data"], key_bytes)
255     _success = verify(_plaintext, _entry["sha256"])
256     return _success, _plaintext
257
258
259 def try_decrypt_image(_entry, range_str, img_width, img_height,
260                     encrypted_raw, pixel_start):
261     """AES-CTR decrypt pixel data and verify against the stored hash.
262
263     Returns (success: bool, decrypted: bytearray).
264     decrypted is always returned so the bitmap updates on every reading change,
265     letting the user see the image snap into focus at the correct sensor level.
266     """
267     key_bytes = derive_key(range_str, b64=False)
268     pixel_data = encrypted_raw[pixel_start: pixel_start + img_width * img_height]
269     _decrypted = bytearray(len(pixel_data))
270     cipher = aesio.AES(key_bytes, aesio.MODE_CTR, IV=_entry["iv"])
271     cipher.decrypt_into(pixel_data, _decrypted)
272     _success = verify(_decrypted, _entry["sha256"])
273     return _success, _decrypted
274
275
276 # =====
277 # CHALLENGE RENDERER
278 # =====
279
280 _image_cache = {}
281
282
283 def load_image_entry(_entry):
284     """Read and parse an encrypted ABMP file. Returns a state dict."""
285     filename = _entry["data"]
286     if filename not in _image_cache:
287         with open(filename, "rb") as f:

```

```

288     raw = bytearray(f.read())
289     _image_cache[filename] = raw
290
291     raw = _image_cache[filename]
292     if raw[0:4] != b"ABMP":
293         raise ValueError(f"Not a valid ABMP file: {filename}")
294
295     img_width, img_height, n_colors = struct.unpack_from("<HHH", raw, 4)
296     palette_start = 10
297     pixel_start = palette_start + n_colors * 3
298
299     palette = Palette(n_colors)
300     for i in range(n_colors):
301         off = palette_start + i * 3
302         r, g, b = raw[off], raw[off + 1], raw[off + 2]
303         palette[i] = (r << 16) | (g << 8) | b
304
305     bitmap = Bitmap(img_width, img_height, n_colors)
306     return {
307         "bitmap": bitmap,
308         "palette": palette,
309         "img_width": img_width,
310         "img_height": img_height,
311         "encrypted_raw": raw,
312         "pixel_start": pixel_start,
313     }
314
315
316     class ChallengeRenderer(Group):
317         """Manages the content_group display for the currently active challenge."""
318
319         def __init__(self):
320             super().__init__()
321             self._active_index = None
322             self.text_widget = None
323             self.image_state = None
324
325             self.secret_message_text = TextBox(
326                 terminalio.FONT,
327                 display.width // 2 - 2,
328                 (display.height) // 2,
329                 align=TextBox.ALIGN_LEFT,
330                 scale=2
331             )
332             self.secret_message_text.anchor_point = (0, 0)
333             self.secret_message_text.anchored_position = (2, 0)
334             self.append(self.secret_message_text)
335             self.secret_message_text.text = "Reading sensor..."
336
337             self.secret_image_tilegrid = None
338
339             self.prompt_text = Label(terminalio.FONT)
340             self.prompt_text.anchor_point = (0, 1.0)
341             self.prompt_text.anchored_position = (2, display.height)
342             self.prompt_text.text = "[ press Boot btn ]"
343             self.prompt_text.hidden = True

```

```

344     self.append(self.prompt_text)
345
346 def setup(self, index):
347     """Prepare content_group for challenge *index* (no-op if already set up)."""
348     if self._active_index == index:
349         return
350
351     if self.secret_image_tilegrid is not None and self.secret_image_tilegrid in self:
352         self.remove(self.secret_image_tilegrid)
353
354     self.text_widget = None
355     self.image_state = None
356     _entry = SEQUENCE[index]
357
358     if _entry["type"] == "text":
359         self.secret_message_text.hidden = False
360         self.secret_message_text.text = "Reading sensor..."
361
362     elif _entry["type"] == "image":
363         state = load_image_entry(_entry)
364         self.image_state = state
365         self.secret_message_text.hidden = True
366
367         self.secret_image_tilegrid = TileGrid(state["bitmap"], pixel_shader=state["palette"])
368         # self.secret_image_tilegrid.transpose_xy = True
369         self.append(self.secret_image_tilegrid)
370
371     self._active_index = index
372
373 def update_image(self, decrypted_pixels):
374     """Blit already-decrypted pixel bytes into the bitmap."""
375     _s = self.image_state
376     bitmaptools.arrayblit(
377         _s["bitmap"], decrypted_pixels,
378         x1=0, y1=0, x2=_s["img_width"], y2=_s["img_height"],
379     )
380
381
382 renderer = ChallengeRenderer()
383 main_group.append(renderer)
384
385
386 # =====
387 # HUD + COMPLETION
388 # =====
389
390 def update_hud(_current_index, _total):
391     """Update the top progress bar label."""
392     label = f"[_current_index + 1]/[_total]"
393     if progress_label.text != label:
394         progress_label.text = label
395
396
397 def show_completion_screen():
398     """Replace challenge content with a 'you win' message and halt."""
399     while len(renderer):

```

```

400     renderer.pop()
401
402     fin = TextBox(
403         terminalio.FONT,
404         display.width // 2,
405         (display.height - 16) // 2,
406         align=TextBox.ALIGN_CENTER,
407         scale=2
408     )
409     fin.anchor_point = (0, 0)
410     fin.anchored_position = (0, 0)
411     fin.text = "All secrets revealed.\nGood job!"
412
413     renderer.append(fin)
414     hud_group.hidden = True
415     while True:
416         pass # halt / wait forever
417
418
419     # =====
420     # MAIN LOOP
421     # =====
422
423     current_index = 0
424     old_range = ""
425     total = len(SEQUENCE)
426     challenge_solved = False # True while waiting for the user to press the button
427
428     last_sensor_read_time = 0
429     cur_cache_key = None
430     while True:
431         entry = SEQUENCE[current_index]
432         now = time.monotonic()
433         if now - last_sensor_read_time > SENSOR_READ_COOLDOWN and not challenge_solved:
434             try:
435                 cur_range = get_range_string_from_sensor(entry)
436             except OSError as e:
437                 print(e, "retrying after cooldown")
438                 cur_range = cur_cache_key.split(" ")[-1]
439                 last_sensor_read_time = now
440             else:
441                 cur_range = cur_cache_key.split(" ")[-1]
442             # Include reading type in cache key so changing challenge boundaries
443             # don't accidentally match a previous challenge's bucket value.
444             cur_cache_key = f"{entry['reading']}:{cur_range}"
445
446             renderer.setup(current_index)
447             update_hud(current_index, total)
448
449             # -----
450             # While a challenge is solved-but-not-yet-confirmed, keep updating
451             # the display with the latest decryption (sensor reading may still drift) but
452             # wait for a button press before advancing.
453             # -----
454             if challenge_solved:
455                 # renderer.show_solved_prompt()

```

```

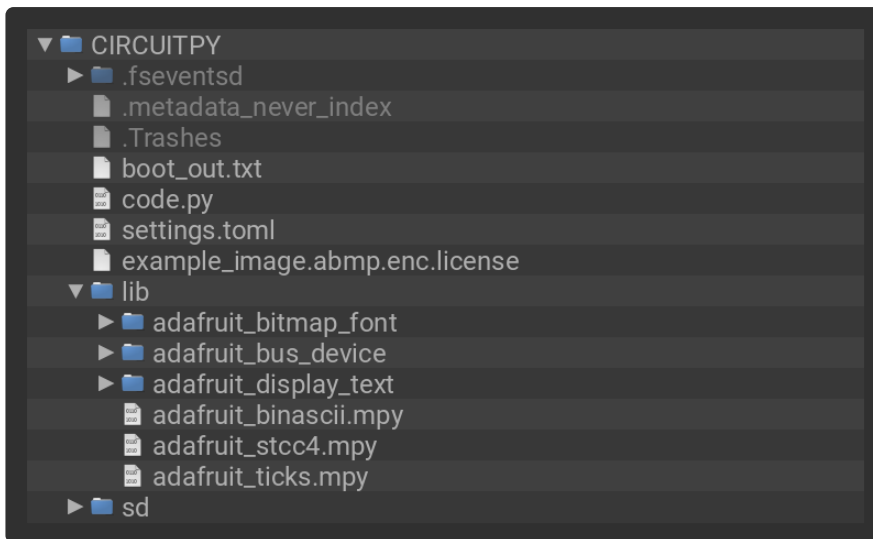
456 btn_cur_val = btn.value
457 if not btn_cur_val and btn_prev_value:
458     print(f"Challenge {current_index + 1} confirmed by button press — advancing.")
459     current_index += 1
460     challenge_solved = False
461     old_range = "" # force re-decrypt immediately on next loop
462
463     if current_index >= total:
464         update_hud(total, total)
465         show_completion_screen()
466     btn_prev_value = btn_cur_val
467     continue # skip normal decrypt logic while waiting for button
468
469 # -----
470 # Normal path: decrypt on every reading-range change and check for a solve.
471 # -----
472 now = time.monotonic()
473 if cur_cache_key != old_range:
474     last_sensor_read_time = now
475     unit = READING_TYPES[entry["reading"]][1]
476     print(f"[{current_index + 1}/{total}] {entry['reading']} range: {cur_range} {unit}")
477     old_range = cur_cache_key
478
479 if entry["type"] == "text":
480     success, plaintext = try_decrypt_text(entry, cur_range)
481     renderer.secret_message_text.text = plaintext
482
483     if success:
484         print(f"Challenge {current_index + 1} SOLVED (text): {plaintext}")
485         cur_reading_label.text = "" + cur_reading_label.text
486         challenge_solved = True
487
488 elif entry["type"] == "image":
489     s = renderer.image_state
490     success, decrypted = try_decrypt_image(
491         entry, cur_range,
492         s["img_width"], s["img_height"],
493         s["encrypted_raw"], s["pixel_start"],
494     )
495     renderer.update_image(decrypted)
496
497     if success:
498         print(f"Challenge {current_index + 1} SOLVED (image)")
499         cur_reading_label.text = "" + cur_reading_label.text
500         challenge_solved = True

```

https://github.com/adafruit/Adafruit_Learning_System_Guides/blob/main/Sensor_Locked_Secrets/stcc4_sht41/code.py

Drive Structure

After copying the files, your drive should look like the listing below. It can contain other files as well, but must contain these at a minimum.



Using Other Sensors

The preceding pages cover versions of the project for the MAX44009, STCC4+SHT41, and GPS Featherwing. If you have a different light sensor, temperature/humidity sensor, or supported GPS receiver, the code can be easily adapted by changing the driver import and sensor initialization. Since all of the Adafruit libraries for the same type of sensors share common property names for their data readings, they are largely interchangeable in these projects. For different GPS breakouts, be sure to update the pins used for initialization according to the module you have.

For example, the following illustrates changing the lux version of the project to use the VCNL4030.

Existing code:

```
1 from adafruit_max44009 import MAX44009
2
3 # ...
4
5 # =====
6 # SENSOR SETUP
7 # =====
8
9 i2c = board.I2C()
10 sensor = MAX44009(i2c)
```

VCNL4030 Adaptation:

```
1 from adafruit_vcnl4030 import VCNL4030
2
3 # ...
4
```

```

5 | # =====
6 | # SENSOR SETUP
7 | # =====
8 |
9 | i2c = board.I2C()
10 | sensor = VCNL4030(i2c)

```

Different Types of Sensors

If you want to venture outside the realm of GPS, lux, temperature, humidity, and CO2, the code can be adapted for other sensor types, but it requires a little more work.

The following sections illustrate changes needed to use a TMAG5273 magnetic sensor.

Change the import and initialization of the driver to use the one for the new sensor.

```

1 | import adafruit_tmag5273
2 |
3 | # ...
4 |
5 | sensor = adafruit_tmag5273.TMAG5273(i2c)

```

Add suitable entries to the `READING_TYPES` dictionary for the new sensor. The magnetic sensor has the property `magnetic` that returns x, y, and z values in a tuple. The attribute name entry in this dictionary is changed from string to tuple so that it can hold a name and a sub-index.

If your sensor has a basic property that returns a single value, then just use a string with the property name like the original project code does.

```

1 | READING_TYPES = {
2 |     # type key : attr info    , unit string
3 |     "magnetic_x": (("magnetic", 0), "uT"),
4 |     "magnetic_y": (("magnetic", 1), "uT"),
5 |     "magnetic_z": (("magnetic", 2), "uT"),
6 | }

```

Update the code inside the `get_range_string_from_sensor()` function to handle the attribute name/sub-indexing scheme.

```

1 | attr_info, _unit = READING_TYPES[_entry["reading"]]
2 | attr_name, subindex = attr_info
3 |
4 | reading = int(getattr(sensor, attr_name)[subindex])

```

Use one of the reading type key values `magnetic_x`, `magnetic_y`, or `magnetic_z` as the value for a custom reading type on the encryptor page.

<https://adafru.it/1aDv>

Text Image

Vigenère Text Encryptor → printable ASCII (32-126)

Reading type

Other (custom)

Custom reading type Precision level Key seed string

magnetic_x 100 800-900

Stored as the "reading" field in the entry. Used directly as the key seed (no range computation).

Plaintext

Magnets!!

Newlines pass through unchanged. Other characters must be in printable ASCII (32-126).

Encrypt

SEQUENCE entry (paste into code.py) copy

```
{
  "type": "text",
  "data": "b6_4HmKeP",
  "sha256":
  "8e22edd5d318d898bf1e792a90fc56380db8e6daeb6be961b7d0b17795a5ac86",
  "reading": "magnetic_x",
  "precision_level": 100,
},
```