A Logger for CircuitPython

Created by Dave Astels

https://learn.adafruit.com/a-logger-for-circuitpython

Last updated on 2023-08-29 04:05:14 PM EDT

Table of Contents

Overview

Have you ever been working on code and needed a view into what was going on as it runs (or tries to)? In many environments you can go into a debugger and poke around. We don't have that ability in CircuitPython, though. If you're like this author, you sprinkle tactical `print` statements as needed.

Afterwards you probably go through and remove them or comment them out. Sometimes you miss some. Sometimes you'd like to leave them in place and be able to turn them on and off. There are times when you'd like to see some debugging information, and other times when you want to be notified of critical errors only.

A logging framework will let you do all that and more.

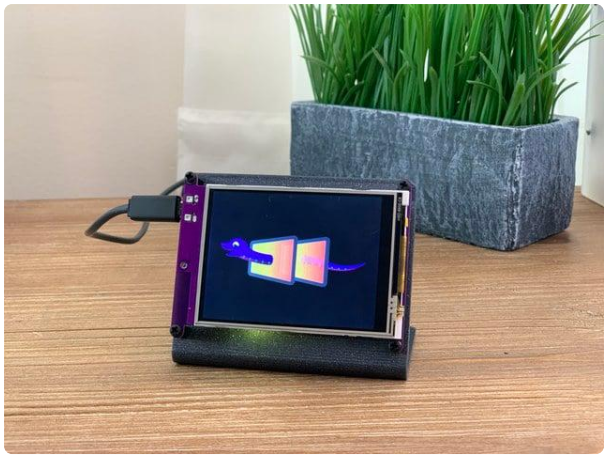
Specifically, the logging framework described in this guide will:

- let you output messages at one of several levels of priority,
- ignore messages below a specific priority,
- automatically add a timestamp to messages,
- provide the string format method support for building messages,
- give you convenience methods for the outputting at standard priority levels,
- control where messages go, and
- make it easy to add new places for messages to go.

This guide will go over the use of the framework, walk through the implementation, and work through an example of adding a new destination capability.

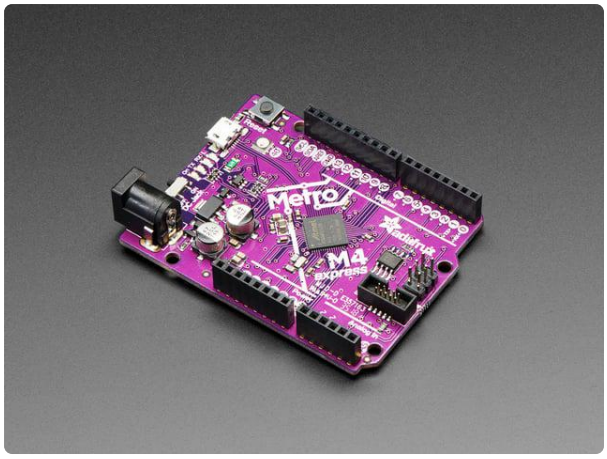
Parts

As this service uses RAM and space for longer programs, this guide will note use on M4 and nRF52840-based boards.



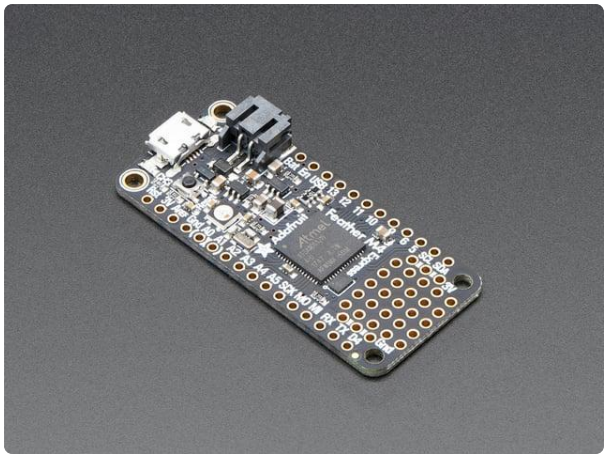
Adafruit PyPortal - CircuitPython Powered Internet Display

PyPortal, our easy-to-use IoT device that allows you to create all the things for the “Internet of Things” in minutes. Make custom touch screen interface...
https://www.adafruit.com/product/4116



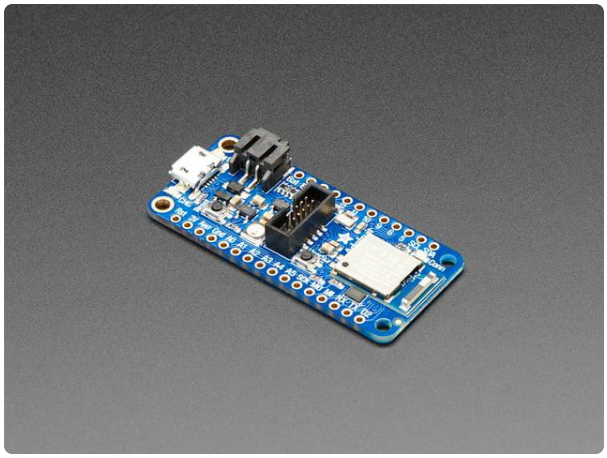
Adafruit Metro M4 feat. Microchip ATSAMD51

Are you ready? Really ready? Cause here comes the fastest, most powerful Metro ever. The Adafruit Metro M4 featuring the Microchip ATSAMD51. This...
https://www.adafruit.com/product/3382



Adafruit Feather M4 Express - Featuring ATSAMD51

It's what you've been waiting for, the Feather M4 Express featuring ATSAMD51. This Feather is fast like a swift, smart like an owl, strong like a ox-bird (it's half ox,...
https://www.adafruit.com/product/3857



Adafruit Feather nRF52840 Express

The Adafruit Feather nRF52840 Express is the new Feather family member with Bluetooth Low Energy and native USB support featuring the nRF52840! It's...
https://www.adafruit.com/product/4062

USB cable - USB A to Micro-B
This here is your standard A to micro-B USB cable, for USB 1.1 or 2.0. Perfect for connecting a PC to your Metro, Feather, Raspberry Pi or other dev-board or...
https://www.adafruit.com/product/592

Using a Logger

First, let's cover how to use a logger in general. The code we're using is very similar to the Python Logging API () so if you've used that, you'll find this familiar

Basic Use

To use the framework, you create a logger and sprinkle logging calls throughout your code at appropriate levels.

```
import adafruit_logging as logging
logger = logging.getLogger('test')

logger.setLevel(logging.ERROR)
logger.info('Info message')
logger.error('Error message')
```

The above example would ignore the info message and output the error one. Messages at any level less than the one set in the `Logger` will be ignored. By default (if you don't set the level) everything will be output. So the output would be:

```
1556.96: ERROR - Error message
```

When you use the log method you can pass in a numeric value, similarly you can set the level of the logger to any numeric value. This gives you the most control over the logger. As an alternative, you can use the 5 defined level values:

- `DEBUG` - 10
- `INFO` - 20
- `WARNING` - 30
- `ERROR` - 40

- `CRITICAL` - 50

When a log message is output, the level gets rounded down. For example, a level of 36 would output as `WARNING`.

To make things easy to use, `Logger` provides a method for each of the levels. As shown above, you can use calls like `logger.error('Error message').`

As mentioned, you can use existing Python formatting strings to build the message:

```
logger.info('Bad value: %d', value)
```

That's pretty much it. You create a logger, add logging statements to your code, and when your code starts up, set the lowest level of messages you want to see.

CircuitPython

Getting Familiar

CircuitPython is a programming language based on Python, one of the fastest growing programming languages in the world. It is specifically designed to simplify experimenting and learning to code on low-cost microcontroller boards. This guide covers the basics:

- Welcome to CircuitPython! ()

Be sure you have the latest CircuitPython for your board loaded onto your board. This should be from no earlier than the end of Feb 2019.

CircuitPython is easiest to use within the Mu Editor. If you haven't previously used Mu, this guide will get you started ().

The logging module will work with any CircuitPython capable board, M0, M4, nRF52840, etc.

Download Library Files

Plug your CircuitPython supported board into your computer via a USB cable. Please be sure the cable is a good power+data cable so the computer can talk to the Feather board.

A new disk should appear in your computer's file explorer/finder called CIRCUITPY. This is the place we'll copy the code and code library. If you can only get a drive named xxxxBOOT, load CircuitPython per the guide above.

Create a new directory on the CIRCUITPY drive named lib.

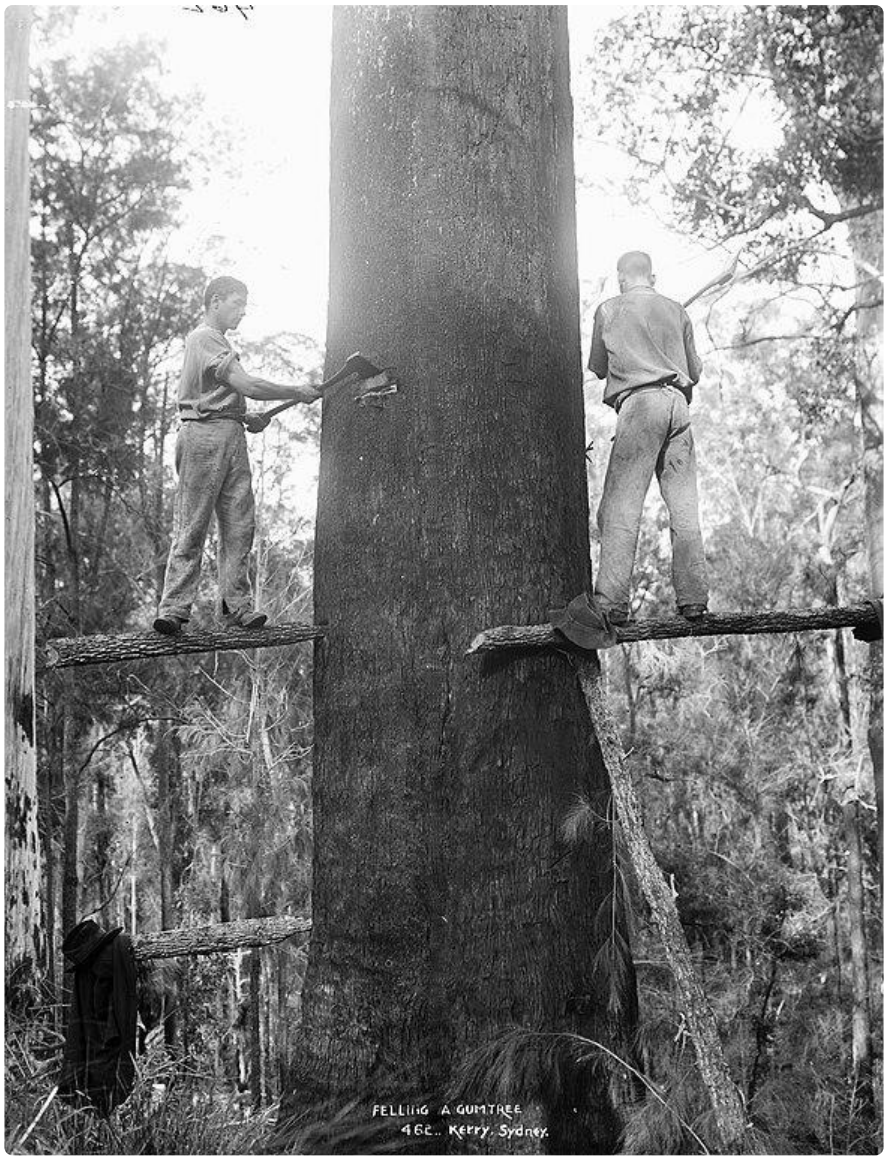
Download the latest CircuitPython driver package to your computer using the green button below. Match the library you get to the version of CircuitPython you are using. Save to your computer's hard drive where you can find it.

Go to GitHub to get the latest CircuitPython library bundle

The logging support is in the adafruit_logger package.

Copy the adafruit_logger package to the /lib directory on your board.

Code Walkthrough



Levels

This module is nice in that it doesn't require any other libraries other than the built-in `time` module.

There is a list that defines the levels: the value and a name. That's used to convert values to names, as well as create a global variable for each level. They can be used directly as, for example, `logging.ERROR`.

```
import time

levels = [(0,  'NOTSET'),
          (10, 'DEBUG'),
          (20, 'INFO'),
          (30, 'WARNING'),
```

```
          (40, 'ERROR'),
          (50, 'CRITICAL')]

for value, name in levels:
    globals()[name] = value

def level_for(value):
    """Convert a numberic level to the most appropriate name.

    :param value: a numeric level

    """
    for i in range(len(LEVELS)):
        if value == LEVELS[i][0]:
            return LEVELS[i][1]
        elif value &lt; LEVELS[i][0]:
            return LEVELS[i-1][1]
    return LEVELS[0][1]
```

Getting a Logger

To get hold of a logger, you use the `getLogger` function. You pass it the name of the logger you want to create or retrieve. This way you can ask for a logger anywhere in your code. Specifying the same name will get you the same logger.

```
logger_cache = dict()

def getLogger(name):
    """Create or retrieve a logger by name.

    :param name: the name of the logger to create/retrieve

    """
    if name not in logger_cache:
        logger_cache[name] = Logger()
    return logger_cache[name]
```

Logger

The core of the module is the `Logger` class. By default loggers use a `PrintHandler` (which we'll look at below) that simply uses `print` to output the messages. To change that to a different handler use the `addHandler` method. The method is called `addHandler` to be closer to CPython's logger. It works slightly differently in that it actually adds an additional handler the the logger rather than replacing it.

`Logger` as a `level` property that allows you to get and set the cuttoff priority level. Messages with a level below the one set are ignored.

Finally, there is the log method that is the core of the class. This takes the level to log at, a format string, and arguments to be inserted into the format string. The `%` operator is used (passing it the supplied arguments) to create the message.

```
class Logger(object):
    """Provide a logging api."""

    def __init__(self):
        """Create an instance.

        :param handler: what to use to output messages. Defaults to a PrintHandler.

        """
        self._level = NOTSET
        self._handler = PrintHandler()

    def setLevel(self, value):
        """Set the logging cuttoff level.

        :param value: the lowest level to output

        """
        self._level = value

    def addHandler(hldr):
        """Sets the handler of this logger to the specified handler.
        *NOTE* this is slightly different from the CPython equivalent which adds
        the handler rather than replaceing it.

        :param hldr: the handler

        """
        self._handler = hldr

    def log(self, level, format_string, *args):
        """Log a message.

        :param level: the priority level at which to log
        :param format_string: the core message string with embedded formatting
directives
        :param args: arguments to ``format_string.format()``, can be empty

        """
        if level &gt;= self._level:
            self._handler.emit(level, format_string % args)
```

Finally, there is a convenience method for logging at each level.

```
    def debug(self, format_string, *args):
        """Log a debug message.

        :param format_string: the core message string with embedded formatting
directives
        :param args: arguments to ``format_string.format()``, can be empty

        """
        self.log(DEBUG, format_string, *args)

    def info(self, format_string, *args):
        """Log a info message.

        :param format_string: the core message string with embedded formatting
directives
        :param args: arguments to ``format_string.format()``, can be empty

        """
        self.log(INFO, format_string, *args)

    def warning(self, format_string, *args):
        """Log a warning message.
```

```
        :param format_string: the core message string with embedded formatting
directives
        :param args: arguments to ``format_string.format()``, can be empty

        """
        self.log(WARNING, format_string, *args)

    def error(self, format_string, *args):
        """Log a error message.

        :param format_string: the core message string with embedded formatting
directives
        :param args: arguments to ``format_string.format()``, can be empty

        """
        self.log(ERROR, format_string, *args)

    def critical(self, format_string, *args):
        """Log a critical message.

        :param format_string: the core message string with embedded formatting
directives
        :param args: arguments to ``format_string.format()``, can be empty

        """
        self.log(CRITICAL, format_string, *args)
```

Handlers

We skipped over that part of the file. And what is that `PrintHandler` we saw in the constructor?

Looking at `Logger`'s `log` method above, we see that the handler object is used to e mit (i.e. send out) the message. The `format_string` and `args` are combined using the `%` operator and the result is sent, along with the level, to the `emit` method of the handler.

Here's the builtin `PrintHandler` along with the `LoggingHandler` abstract base class*.

`LoggingHandler` provides a method, `format`, which takes the level and message to be logged and returns the string to be output, built from a timestamp, the name of the level, and the message.

It also contains a placeholder for the `emit` method which raises a `NotImplementedError` as this method must be implemented by subclasses.

```
class LoggingHandler(object):
    """Abstract logging message handler."""

    def format(self, level, msg):
        """Generate a timestamped message.

        :param level: the logging level
```

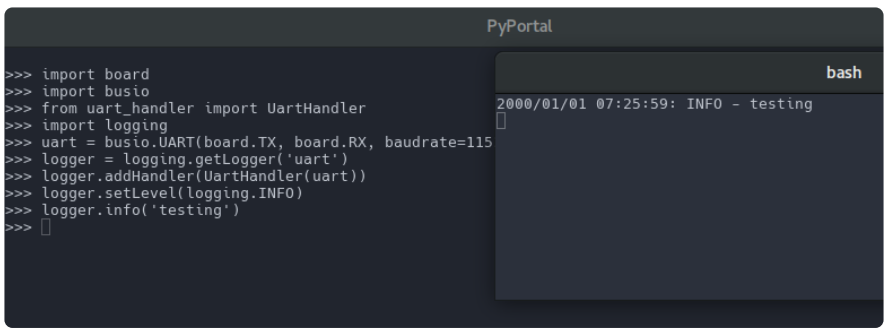
```
        :param msg: the message to log

        """
        return '{0}: {1} - {2}'.format(time.monotonic(), level_for(level), msg)

    def emit(self, level, msg):
        """Send a message where it should go.
        Place holder for subclass implementations.
        """
        raise NotImplementedError()
```

`PrintHandler` subclasses `LoggingHandler` and provides an implementation of `emit` which uses `LoogingHandler`'s `format` method to create the string to be output and prints it. This handler is bundled into the logging module since this is usually what you will need.

```
class PrintHandler(LoggingHandler):
    """Send logging messages to the console by using print."""

    def emit(self, level, msg):
        """Send a message to teh console.

        :param level: the logging level
        :param msg: the message to log

        """
        print(self.format(level, msg))
```

*An abstract base class is not meant to be directly instantiated, rather it is to be subclassed.

Adding Handlers



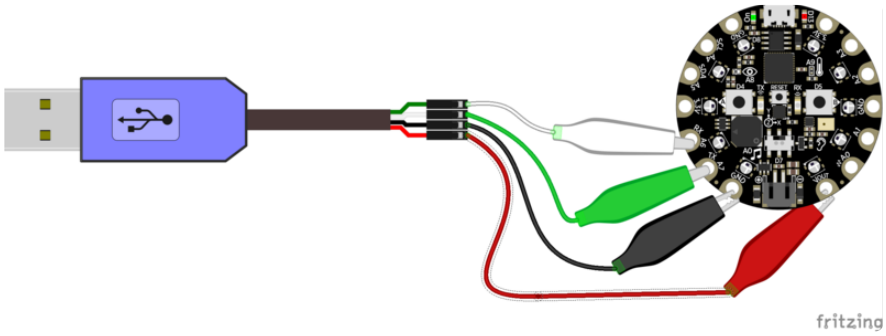
As mentioned earlier, you can write custom handlers to do whatever you need to with the information string to be logged. As an example, you can create a handler to send messages to:

- The serial port (UART)
- A file
- To the Adafruit IO data service
- To a Bluetooth connection

This capability is very helpful when you do not want to mix debug output with output that your code is generating.

The following pages go over the methods of outputting to the differing streams.

Log to UART



With most devboards using the USB connection for the REPL or direct control, you may want to have a secondary USB (or serial) connection - to the same computer or maybe another one. You can also of course use a UART wireless link, XBee, etc. UART is pretty common!

The following code demonstrates logging messages to a board serial (UART) port (usually pin TX):

```
# SPDX-FileCopyrightText: 2018 Dave Astels for Adafruit Industries
#
# SPDX-License-Identifier: MIT

"""
UART based message handler for CircuitPython logging.

Adafruit invests time and resources providing this open source code.
Please support Adafruit and open source hardware by purchasing
products from Adafruit!

Written by Dave Astels for Adafruit Industries
Copyright (c) 2018 Adafruit Industries
Licensed under the MIT license.

All text above must be included in any redistribution.
"""


# Example:
#
# import board
# import busio
# from uart_handler import UartHandler
# import adafruit_logging as logging
#
# uart = busio.UART(board.TX, board.RX, baudrate=115200)
# logger = logging.getLogger('uart')
# logger.addHandler(UartHandler(uart))
# logger.level = logging.INFO
```

```
# logger.info('testing')

from adafruit_logging import Handler

class UartHandler(Handler):
    """Send logging output to a serial port."""

    def __init__(self, uart):
        """Create an instance.

        :param uart: the busio.UART instance to which to write messages
        """
        self._uart = uart

    def format(self, record):
        """Generate a string to log.

        :param record: The record (message object) to be logged
        """
        return super().format(record) + '\r\n'

    def emit(self, record):
        """Generate the message and write it to the UART.

        :param record: The record (message object) to be logged
        """
        self._uart.write(bytes(self.format(record), 'utf-8'))
```

This does a few things.

First, it uses the `UART` instance passed in, giving you the flexibility to use the serial port you want.

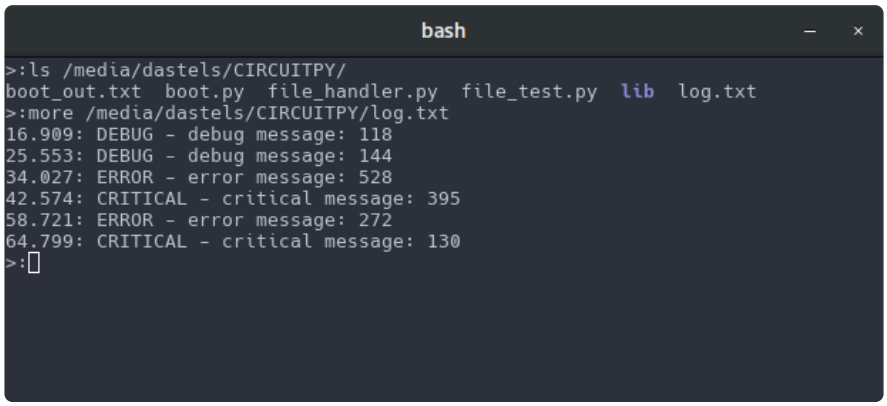
It provides its own `format` method which calls the superclass's `format` to build the output string (that's the `LoggingHandler` class) and appends a newline sequence (a carriage return then a line feed) since `write` doesn't automatically terminate the line the way `print` does.

The `emit` method uses `format` to build the string, converts it to a bytearray and writes the bytes to the UART.

You would use it like in the following example:

```
import board
import busio
from uart_handler import UartHandler
import adafruit_logging as logging

uart = busio.UART(board.TX, board.RX, baudrate=115200)
logger = logging.getLogger('test')
logger.addHandler(UartHandler(uart))
logger.setLevel(logging.INFO)
logger.info('testing')
```

Log to File



A file based handler is similar to the serial port handler, although the output is to a file either on flash (CIRCUITPY drive) or an SD card. If yo an SD card, the SPI bus must be set up to the card interface and the filesystem set.

The handler code is shown below (now part of the logging module):

```
class FileHandler(StreamHandler):
    """File handler for working with log files off of the microcontroller (like
    an SD card)
    :param str filename: The filename of the log file
    :param str mode: Whether to write ('w') or append ('a'); default is to append
    """

    def __init__(self, filename: str, mode: str = "a") -&gt; None:
        # pylint: disable=consider-using-with
        super().__init__(open(filename, mode=mode))

    def close(self) -&gt; None:
        """Closes the file"""
        self.stream.flush()
        self.stream.close()

    def format(self, record: LogRecord) -&gt; str:
        """Generate a string to log
        :param record: The record (message object) to be logged
        """
        return super().format(record) + "\r\n"

    def emit(self, record: LogRecord) -&gt; None:
        """Generate the message and write it to the UART.
        :param record: The record (message object) to be logged
        """
        self.stream.write(self.format(record))
```

You will need to do some extra work to enable your code to write to the file system. The details are covered in this guide ().

Once that's done, you can direct log messages to a file, for example:

```
from file_handler import FileHandler
import adafruit_logging as logging
```

```
l = logging.getLogger('test')
l.addHandler(FileHandler('log.txt'))
l.setLevel(logging.ERROR)
l.error("test")
```

This will result in a file log.txt on the CIRCUITPY drive containing something like:

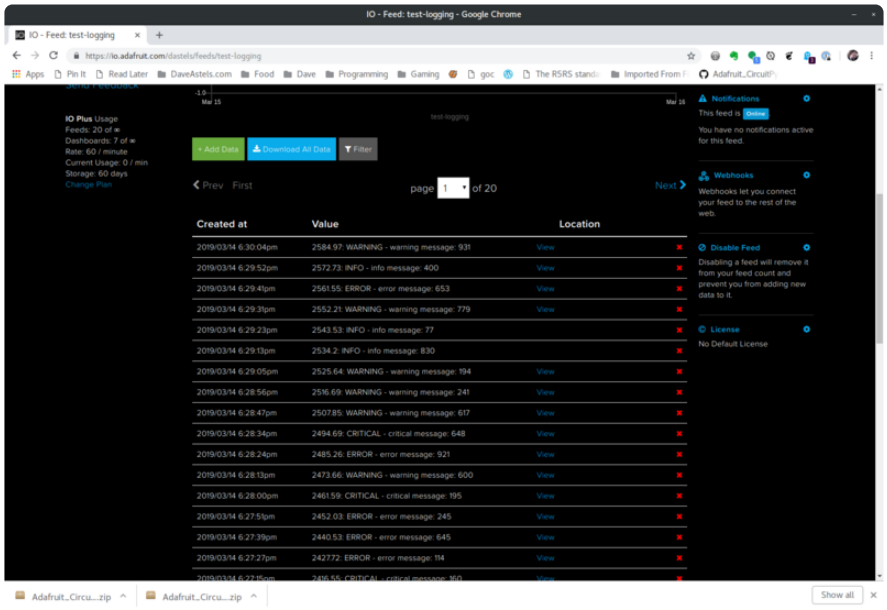
```
1567.13: ERROR - test
```

Log to Adafruit IO

When Internet connectivity is available (usually via WiFi), data may be logged to the Adafruit IO data service.

See this guide to get started with Adafruit IO:

- Welcome to Adafruit IO ()



The following uses a PyPortal (M4 + ESP32) in writing a handler to send log messages to Adafruit IO.

Most of the code is in the constructor to set up the connection to the ESP32 and Adafruit IO. You pass a string to the constructor that is used to create the feed name which is `-logging`.

Line terminators don't need to be added, so we don't need a format method; we can directly use the inherited one.

```
# SPDX-FileCopyrightText: 2018 Dave Astels for Adafruit Industries
#
# SPDX-License-Identifier: MIT

"""
Adafruit IO based message handler for CircuitPython logging.

Adafruit invests time and resources providing this open source code.
Please support Adafruit and open source hardware by purchasing
products from Adafruit!

Written by Dave Astels for Adafruit Industries
Copyright (c) 2018 Adafruit Industries
Licensed under the MIT license.

All text above must be included in any redistribution.
"""

from adafruit_portalbase import PortalBase

# Example:
#
# from aio_handler import AIOHandler
# import adafruit_logging as logging
# l = logging.getLogger('aio')
# # Pass in the device object based on portal_base
# # (Funhouse, PyPortal, MagTag, etc) as the 2nd parameter
# l.addHandler(AIOHandler('test', portal_device))
# l.level = logging.ERROR
# l.error("test")

from adafruit_logging import Handler

class AIOHandler(Handler):

    def __init__(self, name, portal_device):
        """Create an instance."""
        self._log_feed_name=f"{name}-logging"
        if not issubclass(type(portal_device), PortalBase):
            raise TypeError("portal_device must be a PortalBase or subclass of
PortalBase")
        self._portal_device = portal_device


    def emit(self, record):
        """Generate the message and write it to the AIO Feed.

        :param record: The record (message object) to be logged
        """
        self._portal_device.push_to_io(self._log_feed_name, self.format(record))
```

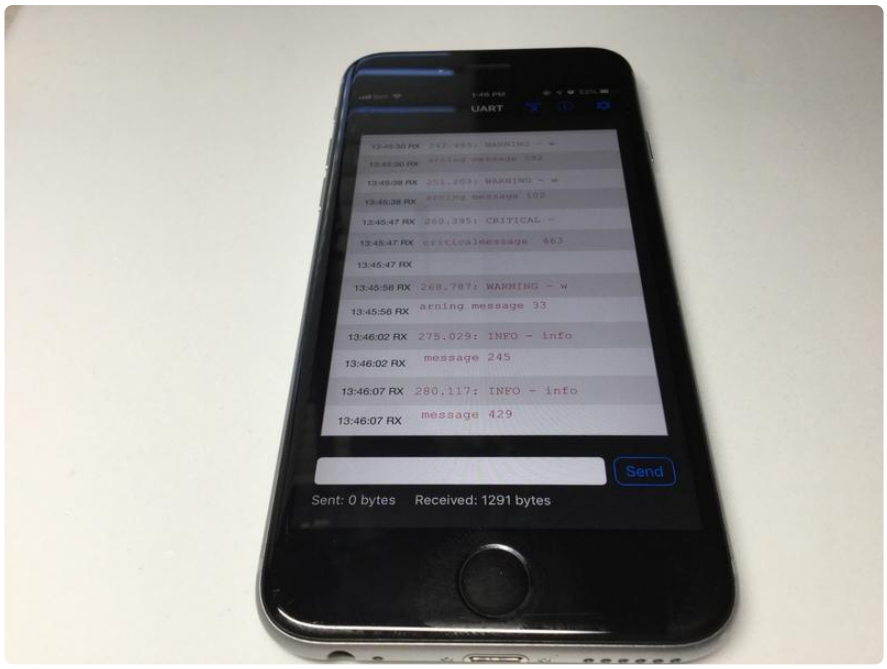
You'll need a secrets.py file to hold your WiFi and Adafruit IO credentials. You will also need the required libraries for your board and an Adafruit IO account. See this guide ( ) for setting it all up on a PyPortal.

The example code to use the above handler on a PyPortal or M4 Express WiFi:

```
from aio_handler import AIOHandler
import adafruit_logging as logging

l = logging.getLogger('aio')
l.addHandler(AIOHandler('test'))
```

```
l.level = logging.ERROR
l.error("test")
```

Log to BLE



If you are using a board that supports BLE, such as the Feather nRF52840, you can write a handler that sends log messages over BLE to, for example, the BlueFruit mobile app. As you can see above, each message is split into 20 character chunks. This is due to the way the low level BLE UART support code operates. Since we use the BLE UART interface, this is very much like the `UARTHandler`.

```
# SPDX-FileCopyrightText: 2019 Dave Astels for Adafruit Industries
#
# SPDX-License-Identifier: MIT

"""
BLE based message handler for CircuitPython logging.

Adafruit invests time and resources providing this open source code.
Please support Adafruit and open source hardware by purchasing
products from Adafruit!

Written by Dave Astels for Adafruit Industries
Copyright (c) 2018 Adafruit Industries
Licensed under the MIT license.

All text above must be included in any redistribution.
"""


from adafruit_logging import Handler
from adafruit_ble.uart import UARTServer

class BLEHandler(Handler):
    """Send logging output to the BLE uart port."""

    def __init__(self):
```

```
        """Create an instance.

        :param uart: the busio.UART instance to which to write messages
        """
        self._advertising_now = False
        self._uart = UARTServer()
        self._uart.start_advertising()

    def format(self, record):
        """Generate a string to log.

        :param record: The record (message object) to be logged
        """
        return super().format(record) + '\r\n'

    def emit(self, record):
        """Generate the message and write it to the UART.

        :param record: The record (message object) to be logged
        """
        while not self._uart.connected:
            pass
        data = bytes(self.format(record), 'utf-8')
        self._uart.write(data)
```

The constructor sets up the BLE UART interface, and starts advertising. This lets devices in the area see it and connect to it. See this guide () for information on using the BlueFruit app. You need to select UART Mode to receive the logging messages from the board.

As with the UART handler, this provides its own `format` method which calls the superclass's `format` to build the output string (that's the `LoggingHandler` class) and appends a newline sequence (a carriage return then a line feed) since `write` doesn't automatically terminate the line the way `print` does.

The `emit` method ensures that there is a live connection, uses `format` to build the string, converts it to a bytearray, and writes the bytes to the BLE UART.

You would use it like in the following example:

```
import board
import busio
from ble_handler import BLEHandler
import adafruit_logging as logging

logger = logging.getLogger('test')
logger.addHandler(BLEHandler())
logger.setLevel(logging.INFO)
logger.info('testing')
```

Testing and Expanding Handlers

Testing handlers

Here's a simple program to test it out a handler. This was used to created the log shown on the Overview page. This shows the Adafruit IO handler but you may change the handler to one of the others.

```
import time
import random
from aio_handler import AIOHandler
import adafruit_logging as logging

l = logging.getLogger('aio')
l.addHandler(AIOHandler('test'))

while True:
    t = random.randint(1, 5)
    if t == 1:
        l.debug("debug message: %d", random.randint(0, 1000))
    elif t == 2:
        l.info("debug message: %d", random.randint(0, 1000))
    elif t == 3:
        l.warning("warning message: %d", random.randint(0, 1000))
    elif t == 4:
        l.error("error message: %d", random.randint(0, 1000))
    elif t == 5:
        l.critical("critical message: %d", random.randint(0, 1000))
    time.sleep(5.0 + (random.random() * 5.0))
```

Getting More Elaborate

A single logger sends it's output to a single place (we've seen console, serial port, and a file), but there's nothing that says you can only have one logger in use. Perhaps you'll want everything logged to a file, and critical errors logged to the console as well. Just create a file based logger and log everything with it, and also have a console logger (using the default PrintLogger) that you use for critical things.

You could even write a custom handler that takes other handlers and routes messages appropriately based on level. For example, logging most messages to a file, but sending critical ones via text or email, or sounding an alarm... it doesn't have to be just outputting strings.