MagTag Lists From Google Spreadsheets

Created by Phillip Burgess

https://learn.adafruit.com/collaborative-spreadsheets-to-magtag

Last updated on 2024-04-06 01:03:50 PM EDT

Table of Contents

Overview

Cloud-connected, programmable devices like Adafruit's **MagTag** make it easy to pull live data from internet sources: tides (https://adafru.it/PcN), space launches (https://adafru.it/P9E), transit schedules (https://adafru.it/Pa2) and more.

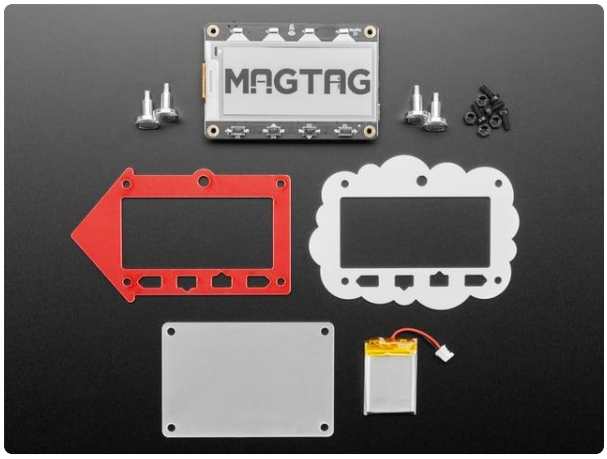
Getting your own stuff out there isn't always so easy though. This often involves server hosting, writing web applications...generally a whole extra layer of knowledge, resources and patience that most of us don't have.

Why reinvent the wheel? We've learned a pretty easy way to do this using **Google Sheets** (https://adafru.it/DCj) — a free, web-based spreadsheet platform. You might already have an account.

One can create and edit lists and reminders easily using a web browser or mobile app...then, combined with some CircuitPython programming, have this information displayed on MagTag. Google Sheets allows multiple people to collaborate on the same document. Or, share just the data feed while the original document is off-limits to others. We'll demonstrate a couple simple examples in this guide...but if you really get to know your way around Google Sheets, it's possible to have it (and thus MagTag) showing dynamic data like stock quotes or days-remaining counters. Some potent magic!

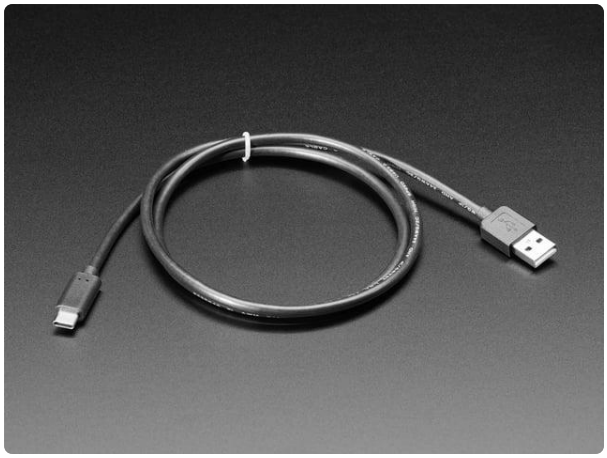
Parts Required

The MagTag starter kit includes an e-ink development board, LiPoly battery and magnetic feet...bring-your-own USB type A to type C cable. Or the individual pieces can be rounded up separately...

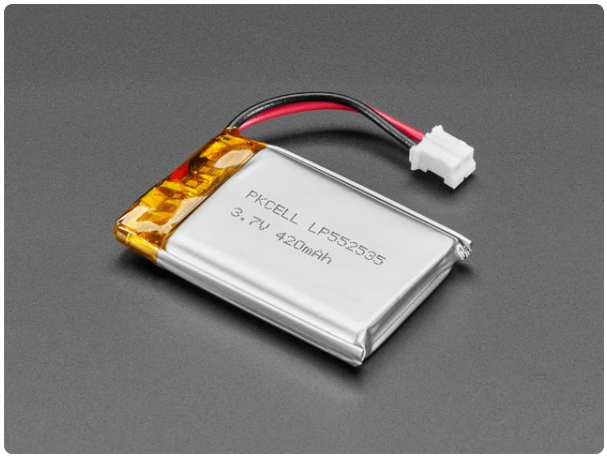


[Adafruit MagTag Starter Kit - ADABOX017 Essentials]
The Adafruit MagTag combines the new ESP32-S2 wireless module and a 2.9" grayscale E-Ink display to make a low-power IoT display that can show data on its screen...
https://www.adafruit.com/product/4819

[Adafruit MagTag - 2.9" Grayscale E-Ink WiFi Display]
The Adafruit MagTag combines the new ESP32-S2 wireless module and a 2.9" grayscale E-Ink display to make a low-power IoT display that can show data on its screen even when power...
https://www.adafruit.com/product/4800



[USB Type A to Type C Cable - approx 1 meter / 3 ft long]
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/4474



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

Also needed:

- **WiFi network** (802.11 b/g/n)
- A desktop or laptop **computer** is required for initial setup: any **text editor** will suffice
- A **Google account** to create and collaborate on cloud-based spreadsheets.

In the next section—specifically the "CircuitPython Internet Test" page—you'll create a **secrets file** to access your wireless network. This is a necessary step, don't just skip ahead. If you've done some WiFi-connected MagTag projects before, you probably already have this file.

Install CircuitPython

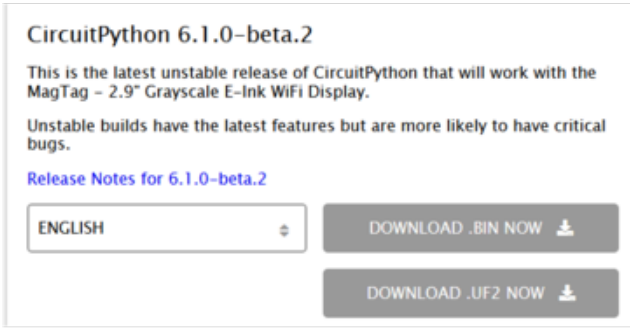
CircuitPython (https://adafru.it/tB7) is a derivative of MicroPython (https://adafru.it/BeZ) designed to simplify experimentation and education on low-cost microcontrollers. It makes it easier than ever to get prototyping by requiring no upfront desktop software downloads. Simply copy and edit files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

Set Up CircuitPython

Follow the steps to get CircuitPython installed on your MagTag.

Download the latest CircuitPython for your board from circuitpython.org

https://adafru.it/OBd



Click the link above and download the latest .BIN and .UF2 file

(depending on how you program the ESP32S2 board you may need one or the other, might as well get both)

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

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

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

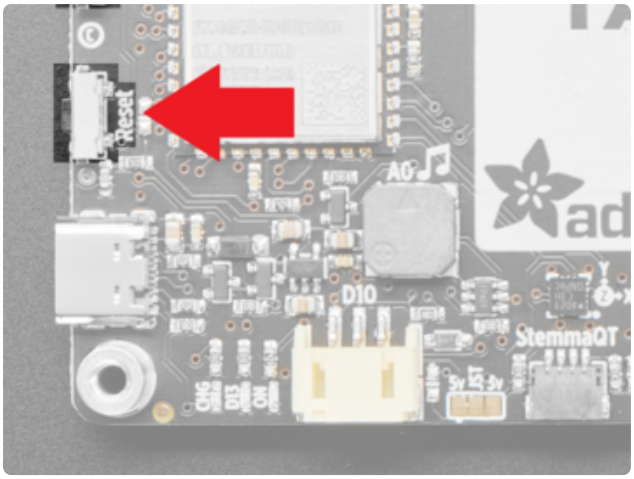
Option 1 - Load with UF2 Bootloader

This is by far the easiest way to load CircuitPython. **However it requires your board has the UF2 bootloader installed. Some early boards do not (we hadn't written UF2 yet!) - in which case you can load using the built in ROM bootloader.**

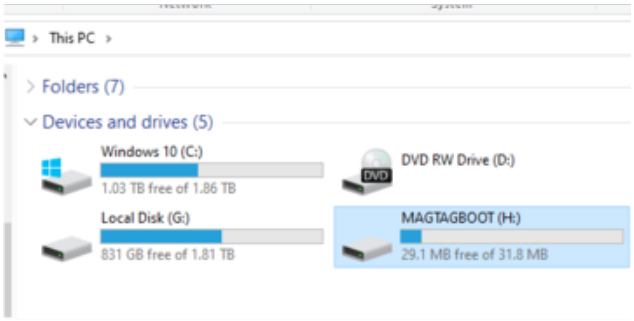
Still, try this first!

Try Launching UF2 Bootloader

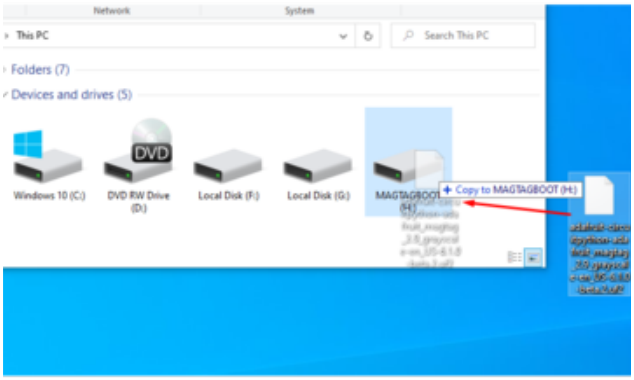
Loading CircuitPython by drag-n-drop UF2 bootloader is the easier way and we recommend it. If you have a MagTag where the front of the board is black, your MagTag came with UF2 already on it.



Launch UF2 by **double-clicking** the Reset button (the one next to the USB C port). You may have to try a few times to get the timing right.

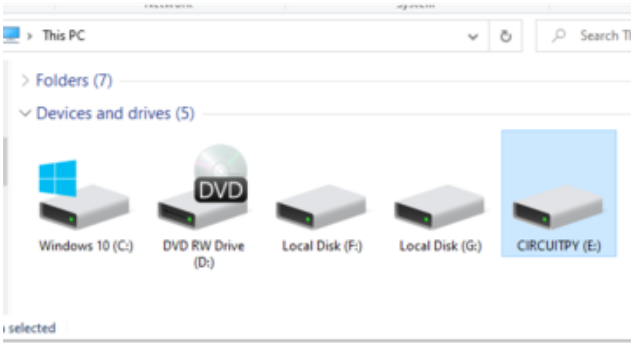


If the UF2 bootloader is installed, you will see a new disk drive appear called **MAGTAGBOOT**



Copy the **UF2** file you downloaded at the first step of this tutorial onto the **MAGTAGBOOT** drive

If you're using Windows and you get an error at the end of the file copy that says **Error from the file copy, Error 0x800701B1: A device which does not exist was specified.** You can ignore this error, the bootloader sometimes disconnects without telling Windows, the install completed just fine and you can continue. If its really annoying, you can also upgrade the bootloader (the latest version of the UF2 bootloader fixes this warning) (https://adafru.it/Pfk)

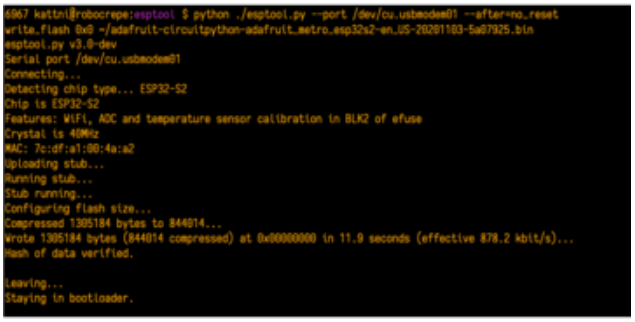


Your board should auto-reset into CircuitPython, or you may need to press reset. A **CIRCUITPY** drive will appear. You're done! Go to the next pages.

Option 2 - Use esptool to load BIN file

If you have an original MagTag with while soldermask on the front, we didn't have UF2 written for the ESP32S2 yet so it will not come with the UF2 bootloader.

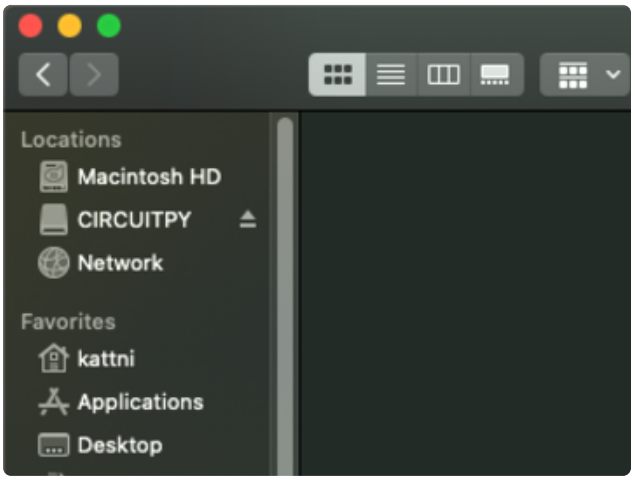
You can upload with **esptool** to the ROM (hardware) bootloader instead!



Follow the initial steps found in the Run esptool and check connection section of the ROM Bootloader page (https://adafru.it/OBc) to verify your environment is set up, your board is successfully connected, and which port it's using.

In the final command to write a binary file to the board, replace the port with your port, and replace "firmware.bin" with the the file you downloaded above.

The output should look something like the output in the image.



Press reset to exit the bootloader.

Your **CIRCUITPY** drive should appear!

You're all set! Go to the next pages.

Option 3 - Use Chrome Browser To Upload BIN file

If for some reason you cannot get esptool to run, you can always try using the Chrome-browser version of esptool we have written. This is handy if you don't have Python on your computer, or something is really weird with your setup that makes esptool not run (which happens sometimes and isn't worth debugging!) You can follow along on the Web Serial ESPTool (https://adafru.it/Pdq) page and either load the UF2 bootloader and then come back to Option 1 on this page, or you can download the CircuitPython BIN file directly using the tool in the same manner as the bootloader.

CircuitPython Internet Test

One of the great things about the ESP32 is the built-in WiFi capabilities. This page covers the basics of getting connected using CircuitPython.

The first thing you need to do is update your **code.py** to the following. Click the **Download Project Bundle** button below to download the necessary libraries and the **code.py** file in a zip file. Extract the contents of the zip file, and copy the **entire lib folder** and the **code.py** file to your **CIRCUITPY** drive.

```
# SPDX-FileCopyrightText: 2020 Brent Rubell for Adafruit Industries
#
# SPDX-License-Identifier: MIT

import os
import ipaddress
import ssl
import wifi
import socketpool
import adafruit_requests

# URLs to fetch from
TEXT_URL = "http://wifitest.adafruit.com/testwifi/index.html"
JSON_QUOTES_URL = "https://www.adafruit.com/api/quotes.php"
JSON_STARS_URL = "https://api.github.com/repos/adafruit/circuitpython"

print("ESP32-S2 WebClient Test")

print(f"My MAC address: {[hex(i) for i in wifi.radio.mac_address]}")

print("Available WiFi networks:")
for network in wifi.radio.start_scanning_networks():
    print("\t%s\t\tRSSI: %d\tChannel: %d" % (str(network.ssid, "utf-8"),
                                              network.rssi, network.channel))
wifi.radio.stop_scanning_networks()

print(f"Connecting to {os.getenv('CIRCUITPY_WIFI_SSID')}")
wifi.radio.connect(os.getenv("CIRCUITPY_WIFI_SSID"),
os.getenv("CIRCUITPY_WIFI_PASSWORD"))
print(f"Connected to {os.getenv('CIRCUITPY_WIFI_SSID')}")
print(f"My IP address: {wifi.radio.ipv4_address}")

ping_ip = ipaddress.IPv4Address("8.8.8.8")
ping = wifi.radio.ping(ip=ping_ip)

# retry once if timed out
if ping is None:
    ping = wifi.radio.ping(ip=ping_ip)

if ping is None:
    print("Couldn't ping 'google.com' successfully")
else:
    # convert s to ms
    print(f"Pinging 'google.com' took: {ping * 1000} ms")

pool = socketpool.SocketPool(wifi.radio)
requests = adafruit_requests.Session(pool, ssl.create_default_context())

print(f"Fetching text from {TEXT_URL}")
response = requests.get(TEXT_URL)
print("-" * 40)
```

```
print(response.text)
print("-" * 40)

print(f"Fetching json from {JSON_QUOTES_URL}")
response = requests.get(JSON_QUOTES_URL)
print("-" * 40)
print(response.json())
print("-" * 40)

print()

print(f"Fetching and parsing json from {JSON_STARS_URL}")
response = requests.get(JSON_STARS_URL)
print("-" * 40)
print(f"CircuitPython GitHub Stars: {response.json()['stargazers_count']}")
print("-" * 40)

print("Done")
```

Your **CIRCUITPY** drive should resemble the following.

To get connected, the next thing you need to do is update the **settings.toml** file.

The settings.toml File

We expect people to share tons of projects as they build CircuitPython WiFi widgets. What we want to avoid is people accidentally sharing their passwords or secret tokens and API keys. So, we designed all our examples to use a **settings.toml** file, that is on your **CIRCUITPY** drive, to hold secret/private/custom data. That way you can share your main project without worrying about accidentally sharing private stuff.

If you have a fresh install of CircuitPython on your board, the initial **settings.toml** file on your **CIRCUITPY** drive is empty.

To get started, you can update the **settings.toml** on your **CIRCUITPY** drive to contain the following code.

```
# SPDX-FileCopyrightText: 2023 Adafruit Industries
#
# SPDX-License-Identifier: MIT
```

```
# This is where you store the credentials necessary for your code.
# The associated demo only requires WiFi, but you can include any
# credentials here, such as Adafruit IO username and key, etc.
CIRCUITPY_WIFI_SSID = "your-wifi-ssid"
CIRCUITPY_WIFI_PASSWORD = "your-wifi-password"
```

This file should contain a series of Python variables, each assigned to a string. Each variable should describe what it represents (say `wifi_ssid`), followed by an = (equals sign), followed by the data in the form of a Python string (such as `"my-wifi-password"` including the quote marks).

At a minimum you'll need to add/update your WiFi SSID and WiFi password, so do that now!

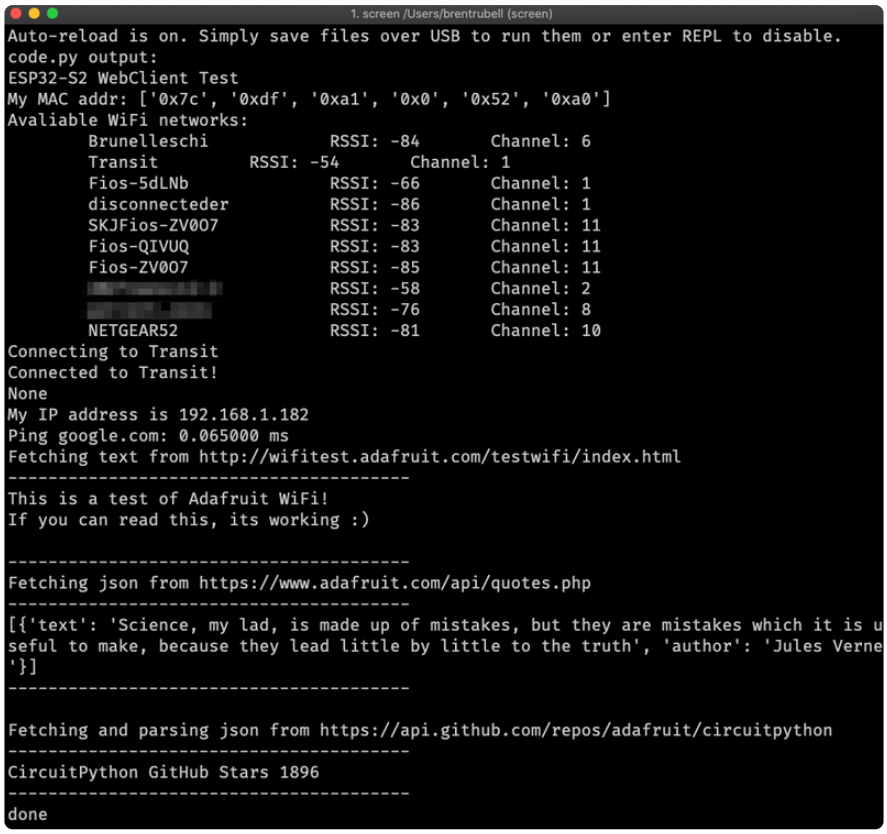
As you make projects you may need more tokens and keys, just add them one line at a time. See for example other tokens such as one for accessing GitHub or the Hackaday API. Other non-secret data like your timezone can also go here.

For the correct time zone string, look at http://worldtimeapi.org/timezones (https://adafru.it/EcP) and remember that if your city is not listed, look for a city in the same time zone, for example Boston, New York, Philadelphia, Washington DC, and Miami are all on the same time as New York.

Of course, don't share your **settings.toml** - keep that out of GitHub, Discord or other project-sharing sites.

Don't share your settings.toml file! It has your passwords and API keys in it!

If you connect to the serial console, you should see something like the following:



In order, the example code...

Checks the ESP32's MAC address.

```
print(f"My MAC address: {[hex(i) for i in wifi.radio.mac_address]}")
```

Performs a scan of all access points and prints out the access point's name (SSID), signal strength (RSSI), and channel.

```
print("Available WiFi networks:")
for network in wifi.radio.start_scanning_networks():
    print("\t%s\t\tRSSI: %d\tChannel: %d" % (str(network.ssid, "utf-8"),
                                              network.rssi, network.channel))
wifi.radio.stop_scanning_networks()
```

Connects to the access point you defined in the **settings.toml** file, and prints out its local IP address.

```
print(f"Connecting to {os.getenv('WIFI_SSID')}")
wifi.radio.connect(os.getenv("WIFI_SSID"), os.getenv("WIFI_PASSWORD"))
print(f"Connected to {os.getenv('WIFI_SSID')}")
print(f"My IP address: {wifi.radio.ipv4_address}")
```

Attempts to ping a Google DNS server to test connectivity. If a ping fails, it returns `None`. Initial pings can sometimes fail for various reasons. So, if the initial ping is successful (`is not None`), it will print the echo speed in ms. If the initial ping fails, it

will try one more time to ping, and then print the returned value. If the second ping fails, it will result in `"Ping google.com: None ms"` being printed to the serial console. Failure to ping does not always indicate a lack of connectivity, so the code will continue to run.

```
ping_ip = ipaddress.IPv4Address("8.8.8.8")
ping = wifi.radio.ping(ip=ping_ip) * 1000
if ping is not None:
    print(f"Ping google.com: {ping} ms")
else:
    ping = wifi.radio.ping(ip=ping_ip)
    print(f"Ping google.com: {ping} ms")
```

The code creates a socketpool using the wifi radio's available sockets. This is performed so we don't need to re-use sockets. Then, it initializes a a new instance of the requests (https://adafru.it/E9o) interface - which makes getting data from the internet really really easy.

```
pool = socketpool.SocketPool(wifi.radio)
requests = adafruit_requests.Session(pool, ssl.create_default_context())
```

To read in plain-text from a web URL, call `requests.get` - you may pass in either a http, or a https url for SSL connectivity.

```
print(f"Fetching text from {TEXT_URL}")
response = requests.get(TEXT_URL)
print("-" * 40)
print(response.text)
print("-" * 40)
```

Requests can also display a JSON-formatted response from a web URL using a call to `requests.get`.

```
print(f"Fetching json from {JSON_QUOTES_URL}")
response = requests.get(JSON_QUOTES_URL)
print("-" * 40)
print(response.json())
print("-" * 40)
```

Finally, you can fetch and parse a JSON URL using `requests.get`. This code snippet obtains the `stargazers_count` field from a call to the GitHub API.

```
print(f"Fetching and parsing json from {JSON_STARS_URL}")
response = requests.get(JSON_STARS_URL)
print("-" * 40)
print(f"CircuitPython GitHub Stars: {response.json()['stargazers_count']}")
print("-" * 40)
```

OK you now have your ESP32 board set up with a proper **settings.toml** file and can connect over the Internet. If not, check that your **settings.toml** file has the right SSID and password and retrace your steps until you get the Internet connectivity working!

Getting The Date & Time

A very common need for projects is to know the current date and time. Especially when you want to deep sleep until an event, or you want to change your display based on what day, time, date, etc. it is

Determining the correct local time is really really hard. There are various time zones, Daylight Savings dates, leap seconds, etc. Trying to get NTP time and then back-calculating what the local time is, is extraordinarily hard on a microcontroller just isn't worth the effort and it will get out of sync as laws change anyways.

For that reason, we have the free adafruit.io time service. **Free for anyone with a free adafruit.io account.** You do need an account because we have to keep accidentally mis-programmed-board from overwhelming adafruit.io and lock them out temporarily. Again, it's free!

There are other services like WorldTimeAPI, but we don't use those for our guides because they are nice people and we don't want to accidentally overload their site. Also, there's a chance it may eventually go down or also require an account.

Step 1) Make an Adafruit account

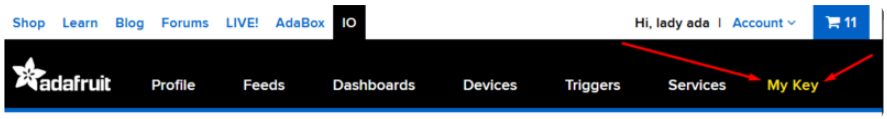
It's free! Visit https://accounts.adafruit.com/ (https://adafru.it/dyy) to register and make an account if you do not already have one

Step 2) Sign into Adafruit IO

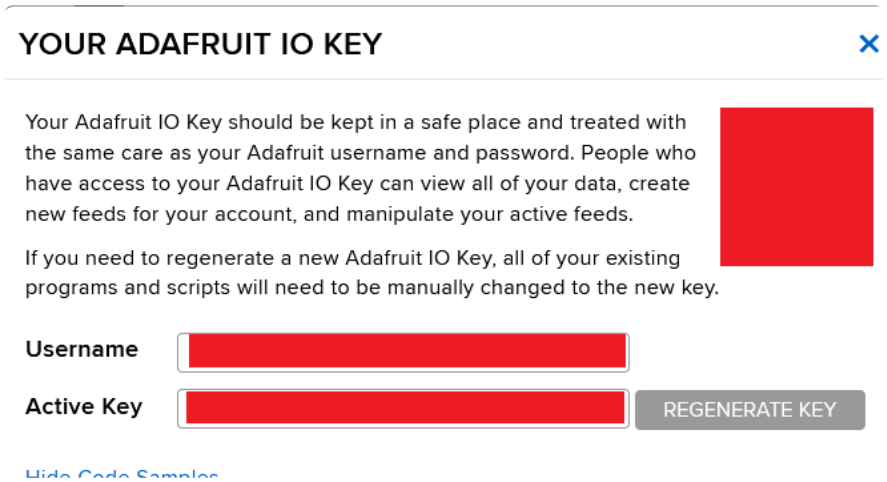
Head over to io.adafruit.com (https://adafru.it/fsU) and click **Sign In** to log into IO using your Adafruit account. It's free and fast to join.

Step 3) Get your Adafruit IO Key

Click on **My Key** in the top bar



You will get a popup with your **Username** and **Key** (In this screenshot, we've covered it with red blocks)



Go to the **settings.toml** file on your CIRCUITPY drive and add three lines for `AIO_USERNAME`, `ADAFRUIT_AIO_KEY` and `TIMEZONE` so you get something like the following:

```
# This file is where you keep secret settings, passwords, and tokens!
# If you put them in the code you risk committing that info or sharing it

CIRCUITPY_WIFI_SSID = "your-wifi-ssid"
CIRCUITPY_WIFI_PASSWORD = "your-wifi-password"
ADAFRUIT_AIO_USERAME = "your-adafruit-io-username"
ADAFTUIT_AIO_KEY = "your-adafruit-io-key"
# Timezone names from http://worldtimeapi.org/timezones
TIMEZONE="America/New_York"
```

The timezone is optional, if you don't have that entry, adafruit.io will guess your timezone based on geographic IP address lookup. You can visit http://worldtimeapi.org/timezones (https://adafru.it/EcP) to see all the time zones available (even though we do not use Worldtime for time-keeping, we do use the same time zone table).

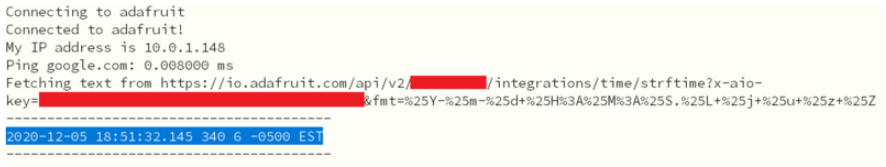
Step 4) Upload Test Python Code

This code is like the Internet Test code from before, but this time it will connect to adafruit.io and get the local time

```
import ipaddress
import os
import ssl
import wifi
```

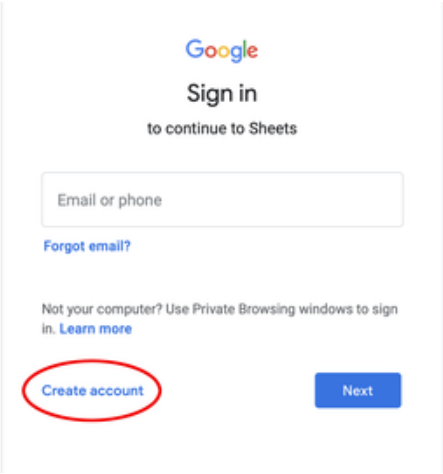
```
import socketpool
import adafruit_requests
import secrets


# Get our username, key and desired timezone
ssid = os.getenv("CIRCUITPY_WIFI_SSID")
password = os.getenv("CIRCUITPY_WIFI_PASSWORD")
aio_username = os.getenv("ADAFRUIT_AIO_USERNAME")
aio_key = os.getenv("ADAFRUIT_AIO_KEY")
timezone = os.getenv("TIMEZONE")
TIME_URL = f"https://io.adafruit.com/api/v2/{aio_username}/integrations/time/
strftime?x-aio-key={aio_key}&amp;tz={timezone}")
TIME_URL += "&amp;fmt=%25Y-%25m-%25d+%25H%3A%25M%3A%25S.%25L+%25j+%25u+%25z+%25Z"

print("ESP32-S2 Adafruit IO Time test")

print("My MAC addr:", [hex(i) for i in wifi.radio.mac_address])

print("Available WiFi networks:")
for network in wifi.radio.start_scanning_networks():
    print("\t%s\t\tRSSI: %d\tChannel: %d" % (str(network.ssid, "utf-8"),
            network.rssi, network.channel))
wifi.radio.stop_scanning_networks()

print("Connecting to", ssid)
wifi.radio.connect(ssid, password)
print(f"Connected to {ssid}!")
print("My IP address is", wifi.radio.ipv4_address)

ipv4 = ipaddress.ip_address("8.8.4.4")
print("Ping google.com:", wifi.radio.ping(ipv4), "ms")

pool = socketpool.SocketPool(wifi.radio)
requests = adafruit_requests.Session(pool, ssl.create_default_context())

print("Fetching text from", TIME_URL)
response = requests.get(TIME_URL)
print("-" * 40)
print(response.text)
print("-" * 40)
```

After running this, you will see something like the below text. We have blocked out the part with the secret username and key data!

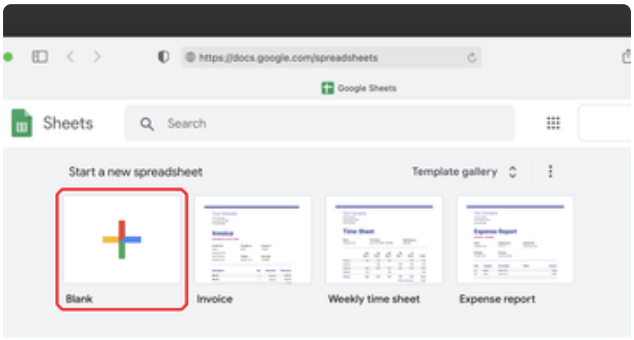


Note at the end you will get the date, time, and your timezone! If so, you have correctly configured your **settings.toml** and can continue to the next steps!

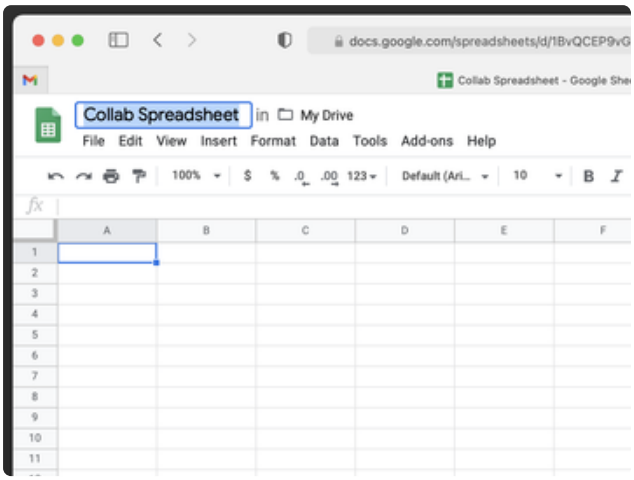
Spreadsheet Setup



Start at https://docs.google.com/spreadsheets **(https://adafru.it/DCj)** and **sign in**...or, if you don't already have a **Google account**, there's an option to **create** one.



Create a new **blank** sheet. Just the basic plain one.



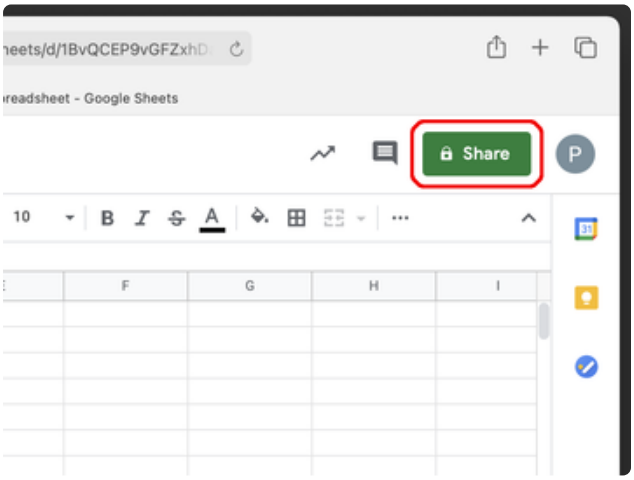
Be a dearie and double-click "Untitled spreadsheet," give it a descriptive name.

Two things to keep in mind:

1. Keep the spreadsheet design **small and simple**. A few dozen cells, tops. Text, numbers and dates are fine. No graphics or colors or merged cells. We're just using this spreadsheet to hold some short notes, not doing serious data analysis

or structuring. Formulas and functions can be used, as long as the output is text, numbers or dates.

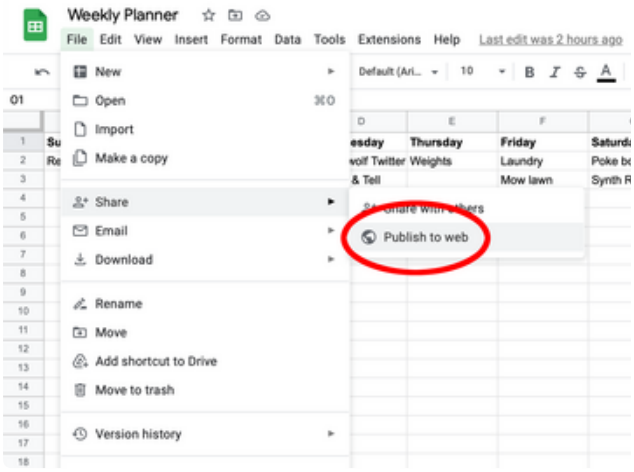
2. Later, on MagTag, there will be some accompanying **CircuitPython** code that **you** will write, and it needs to know how the table is formatted...the whole spreadsheet doesn't simply appear there. We'll walk through some **examples** on pages that follow. Each one's a little different.

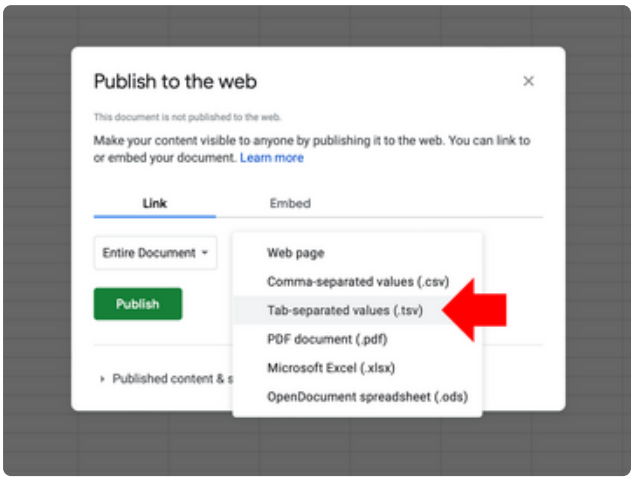


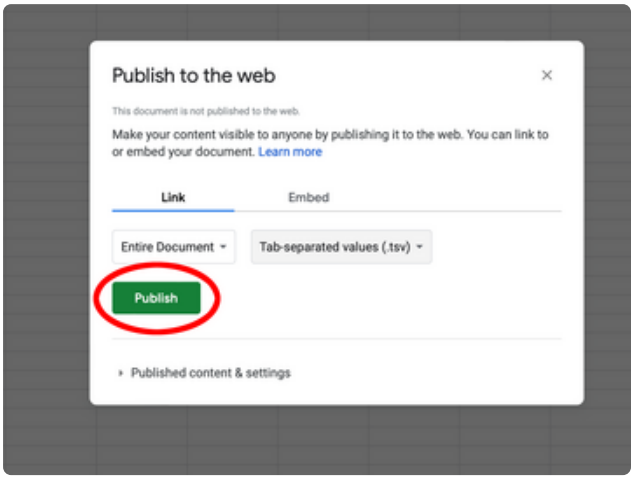
If you'll be **collaborating** with others, click the **Share** button at top-right. You can then add people by email address (they'll also need Google accounts). Everyone can then add to or edit the spreadsheet, even at the same time.

If it's **just you** editing the data, you can **skip** this step.

Publish to Web



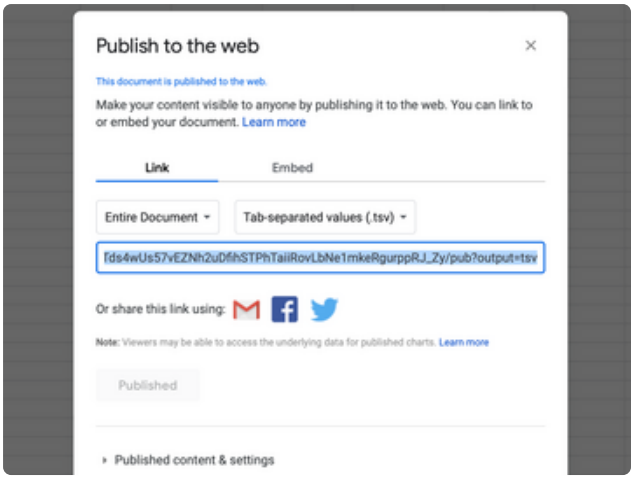




From Google Sheets' **File** menu (not the browser's File menu), select **"Share,"** which rolls over to include "**Publish to web**."

A dialog box pops up with various settings and menus. Most can be left at their defaults, but the second menu selects the published format: change this from the default "Web page" to "**Tab-separated values (.tsv)**" then click the "**Publish**" button. Confirm "**OK**" when asked.

The dialog box expands a bit now, and includes a long URL for others to access the spreadsheet. **Copy** that URL string to the clipboard, we'll paste it into some CircuitPython code later. Double check that this ends with "output=tsv". If not, change the format in the menu above this field.



Never publish sheets containing sensitive information. Anyone with the link is able to read (but not modify) what's there.

You only need to set this up once. Any changes to the original spreadsheet are automatically re-published using the same link you just put together.

Next Steps

Now we'll look at what that the .tsv format is and write some **CircuitPython** code to convert it into something **readable** on the MagTag's display...

Spreadsheet Parsing in CircuitPython

On this page, we'll show some fragments of CircuitPython code, but these are not yet complete programs. We'll bring all the pieces together on the example pages!

On the previous page we created a spreadsheet, published it to the web in .tsv format, and created a link so others (or our MagTag) can access the data.

TSV stands for **Tab-Separated Values**, a simple and fairly standard way for programs to exchange spreadsheet data using plain text. If you've done much programming, you may have encountered .CSV or Comma-Separated Value files. Nearly identical, just using a different separator...commas are easier for human-edited files, while tab separators make it easier for code when spreadsheet cells themselves may contain comma or quote characters.

You'll only need to go through this process once, when designing your CircuitPython code, not every time you edit the spreadsheet data. The exception is if you restructure the spreadsheet, changing the fundamental data organization, in which case you'll need to revisit these steps and think about your code.

For brevity's sake, the code fragments that follow assume...

- Your MagTag board is already running CircuitPython (explained on the "Install CircuitPython" page)
- The **secrets.py** file is configured with your WiFi network credentials (explained on the "CircuitPython Internet Test" page)
- Any additional libraries required by the code are installed and imported (below we show just the minimum, but the examples contain complete programs)

CircuitPython code to access the Google .TSV data requires:

- A link to the published spreadsheet (**TSV_URL** in the code fragment below, but change the string to your published Sheet link URL)
- Instantiating a MagTag object (**MAGTAG** below)
- Connecting to the wireless network
- Calling **MAGTAG.network.fetch(TSV_URL)** and verifying the response. If successful (status code 200), **TSV_DATA** is then a Python string containing the whole spreadsheet structure:

```
from adafruit_magtag.magtag import MagTag

TSV_URL = 'https://YOUR_PUBLISHED_SHEET_LINK'

MAGTAG = MagTag()
MAGTAG.network.connect()

RESPONSE = MAGTAG.network.fetch(TSV_URL)
if RESPONSE.status_code == 200:
    TSV_DATA = RESPONSE.text
```

This is a bit different from some other MagTag (or PyPortal or MatrixPortal) projects, where the URL is passed to the `MagTag()` constructor and `fetch()` simply returns one or more values — a stock price, a temperature, a date and time. That works when the source data is in known fixed format...but here we may need to sift through an indeterminate number of cells in our spreadsheet. We need it all.

The spreadsheet data arrives as one huge string, but this is easily split into multiple separate lines, **one per row** of the original spreadsheet, with Python's built-in split() function, passing `'\r\n'` as a separator (carriage return + line feed).

```
# Split text response into separate lines
LINES = TSV_DATA.split('\r\n')
```

LINES[0] is now the first row of data, LINES[1] the second, and so forth. You can iterate through these like any Python list.

Each line is a string. These in turn can be broken out into individual cells, **one per column**, using split() again, with a tab character (`'\t'`) this time:

```
for line in LINES:
    cells = line.split('\t')
```

It is up to you then to pick through rows and columns to extract the data you need and display it as you want. The examples on the following pages show different things that might be done with this information.

Also helpful to know:

- When splitting into rows or columns, the resulting lists are indexed from **0** in CircuitPython...different from the original spreadsheet, where rows start with **1** and columns with **A**.
- Empty cells will return an empty string (`""`). Your code may need to watch for this and act accordingly to avoid errors.
- The examples use try/except to catch errors. That's a good and neighborly thing to have in a finished program. During development though, you might want to just let the code throw errors, in case you try accessing items out of range, etc.

Installing Examples

The button below downloads a ZIP file of **images and fonts** for the examples. The "Download Project Bundle" links on subsequent pages will fetch the necessary code and libraries, but not these files. You'll need both to piece together a working example.

Download ZIP file for MagTag Google Sheets examples

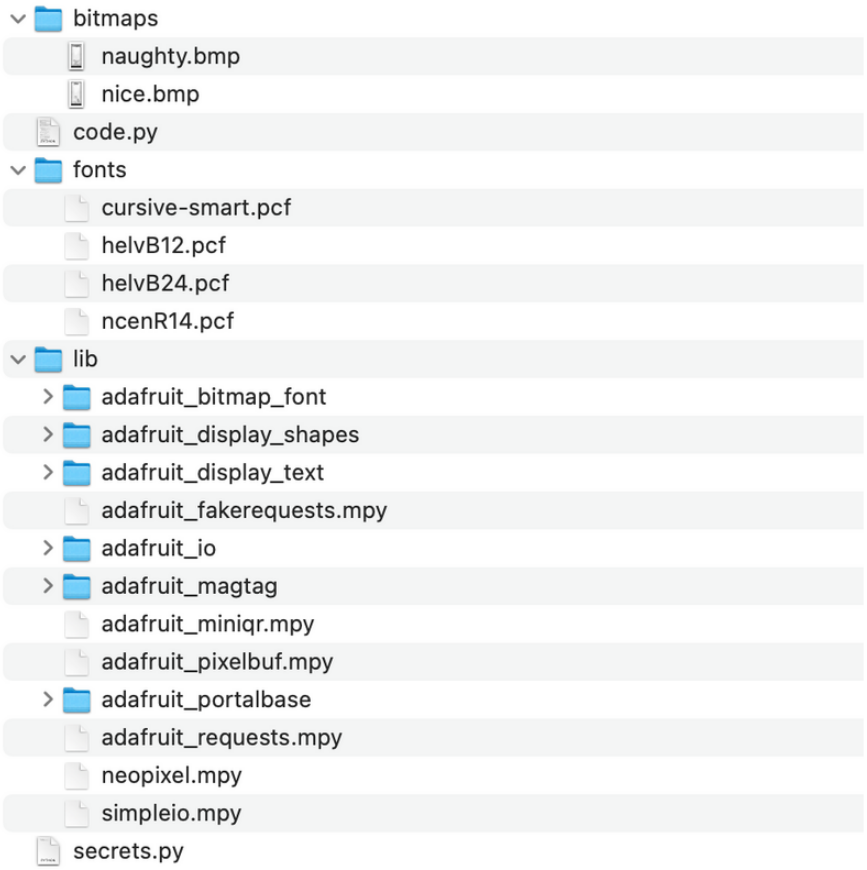
https://adafru.it/19F3

If you're having difficulty running this example, it could be because your MagTag CircuitPython firmware or library needs to be upgraded! Please be sure to follow https://learn.adafruit.com/adafruit-magtag/circuitpython to install the latest CircuitPython firmware and then also replace/update ALL the MagTag-specific libraries mentioned here https://learn.adafruit.com/adafruit-magtag/circuitpython-libraries-2

This is **not** a complete package. You'll still need a few things...

- You must **create and publish the spreadsheets** as shown on each example page. Don't fret, they're simple ones.
- Prerequisite **CircuitPython libraries** must be installed (see below).
- **secrets.py** file (not a part of the project files) must be configured with your **WiFi network** and **Adafruit IO** credentials (as shown on the "CircuitPython Internet" Test page) and time zone ("Getting the Date & Time" page).
- The CircuitPython examples have **descriptive filenames**, but the active file should be renamed **code.py** when copied to the **CIRCUITPY** drive, otherwise it won't run.
- Your **spreadsheet links** must be pasted into the example code.

Here's a map of all this project's required images, fonts and code on the **CIRCUITPY** drive. Remember that nothing will happen until one of the examples is renamed **code.py**:



If you run out of space when copying items to CIRCUITPY: make a **backup** of any files currently on that drive, then delete files that aren't related to this project to free up space.

After setting up any of the examples, tap the board's RESET button. Occasionally the WiFi connection has some lingering data buffered from prior code, and this can cause the examples to fail with an error the first time they're run. RESET makes sure everything starts clean and fresh.

Font Licenses

Example: Naughty or Nice?

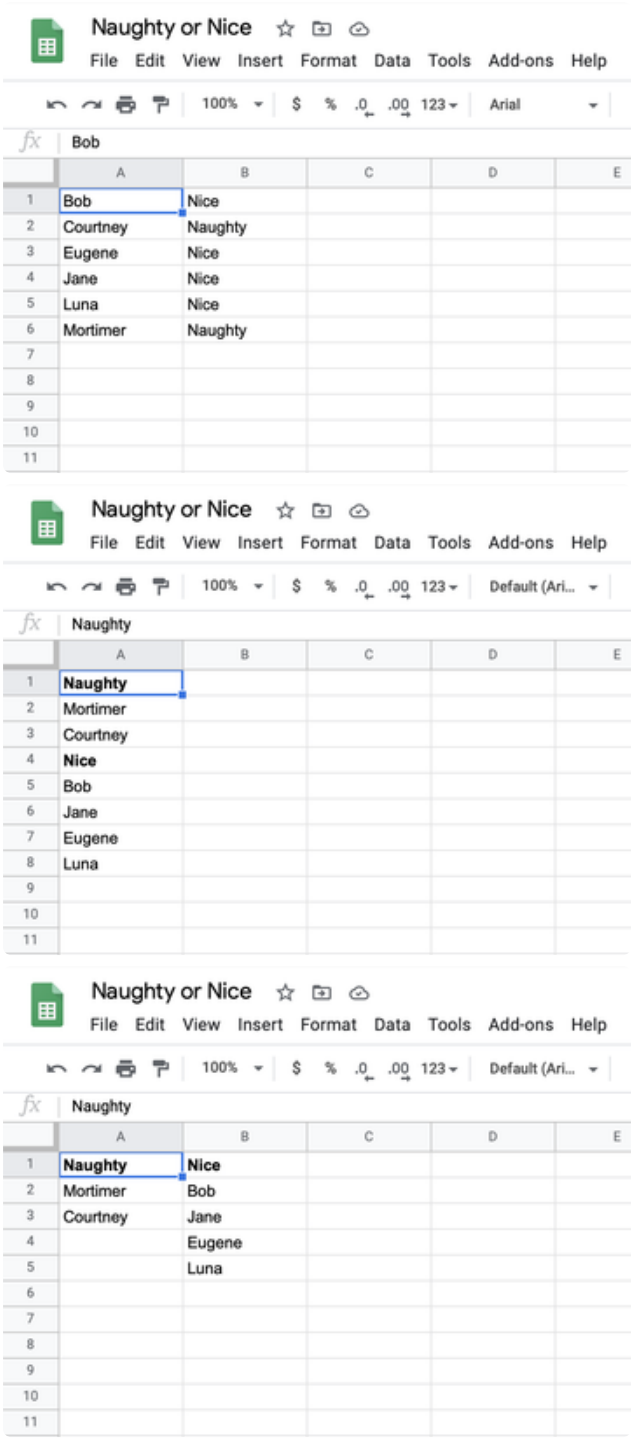
St. Nick and **Krampus** work as a team. **Gifts** for the **nice** kids, **punishment** for the **naughty**. It's important that they **coordinate their efforts**. Can't bring a kid the latest game console, only to be stuffed in a basket, dragged to the underworld and devoured later the same evening...that would just be super awkward, you know?

The two of them can **collaborate remotely** from their respective workshop or dank cave on a **single spreadsheet** that organizes kids into two groups: naughty and nice. St. Nick and Krampus each have **their own separate MagTag board**, which lists **only the names relevant to their particular task.**

You can test this out with just **one MagTag**...decide if you're on **Team Santa** or **Team Krampus**.

Making the Spreadsheet

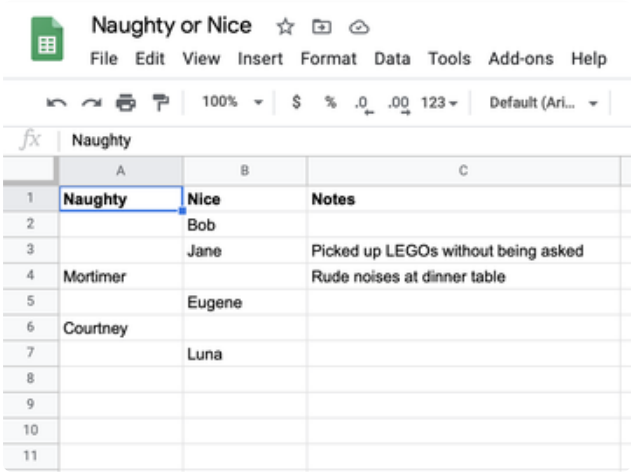
How should we structure this? The most important idea here is that **there's no One Right Way™ to do it**. Find a layout that collaborators find easy to manage, then **write the CircuitPython code to work with that layout**.



We could make one column a list of names, and a second column with a "naughty" or "nice" string for each. But... typos in the second column might be a problem, and all those strings make the data bulkier than needed.

Or we could make one vertical list of names, separated by "naughty" or "nice" spreadsheet cells...the CircuitPython code could look for those strings and make the switch at that point.

Or...since a spreadsheet is two-dimensional...we could make two columns, one for "naughty" and another for "nice." This is good and compact. We'll use something very similar for the next example.



For this one though...we'll start with the "naughty" and "nice" columns, and each row gets just one name, placed in one column or the other. A "sparse" spreadsheet.

A third column, "notes," has been added. Maybe a few kids are right on the naughty/nice fence, and this way St. Nick and Krampus can leave notes to each other to justify why they chose one way or the other.

The notes are only used while working on the spreadsheet though. The CircuitPython code we'll write in a moment ignores that column, each MagTag will only display its respective curated name list.

Remember, you can use most any layout you like, so long as the CircuitPython code is written to match. If following along with this example, use the sparse 3-column format above.

Publish the Spreadsheet

Follow the directions on the "Spreadsheet Setup" page to get the data ready for MagTag. Publish to web, then copy the document's URL link. You only need to do this part once, any document changes will use the same link.

CircuitPython Code

As mentioned on the "Spreadsheet Parsing" page, the **spreadsheet layout** and our **CircuitPython code** need to work hand-in-hand.

Here's the Naughty-or-Nice CircuitPython code (**naughty_nice.py**). The vital bits were already explained on the prior page, and much of it is recycled in the next example. So it's like 90% boilerplate, but below the code a couple of "tricks" are explained.

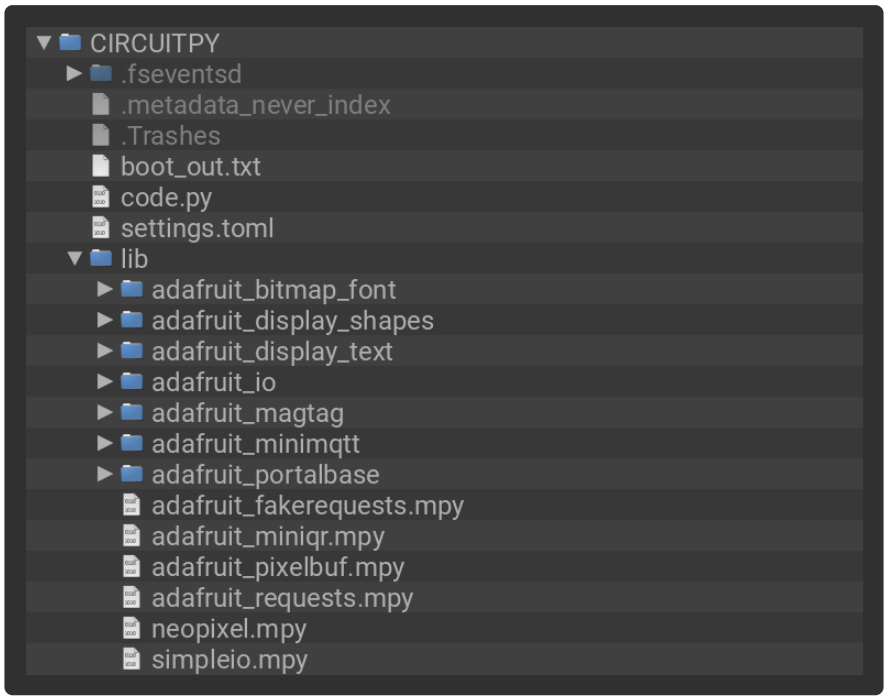
Update TSV_URL to point to your published spreadsheet, and rename the file to "code.py" to run automatically on the CIRCUITPY drive.

Installing Project Code

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

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

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



If you're having difficulty running this example, it could be because your MagTag CircuitPython firmware or library needs to be upgraded! Please be sure to follow https://learn.adafruit.com/adafruit-magtag/circuitpython to install the latest CircuitPython firmware and then also replace/update ALL the MagTag-specific libraries mentioned here https://learn.adafruit.com/adafruit-magtag/circuitpython-libraries-2

```
# SPDX-FileCopyrightText: 2020 Phillip Burgess for Adafruit Industries
#
# SPDX-License-Identifier: MIT

"""
Google Sheets to MagTag example: Naughty or Nice?
```

```
Gets tab-separated-value (TSV) spreadsheet from Google, displays names from
one column or other.
"Smart cursive" font by Thomas A. Fine, helvB12 from Xorg fonts.
"""

# pylint: disable=import-error, line-too-long
import time
import rtc
from adafruit_display_shapes.rect import Rect
from adafruit_magtag.magtag import MagTag


# CONFIGURABLE SETTINGS and ONE-TIME INITIALIZATION -----------------------

TSV_URL = 'https://docs.google.com/spreadsheets/d/e/
2PACX-1vTA8pXQodbEiz5idGT21YkL1Vy8waW0aAHM1uX7D4TqBq6DrUU8qXVlON1QVaWSlmoC3OBL4Iokyiyy/
pub?output=tsv'
NICE = True        # Use 'True' for nice list, 'False' for naughty
TWELVE_HOUR = True # If set, show 12-hour vs 24-hour (e.g. 3:00 vs 15:00)
DD_MM = False      # If set, show DD/MM instead of MM/DD dates

MAGTAG = MagTag(rotation=0) # Portrait (vertical) display


# SOME UTILITY FUNCTIONS --------------------------------------------------

def hh_mm(time_struct, twelve_hour=True):
    """ Given a time.struct_time, return a string as H:MM or HH:MM, either
        12- or 24-hour style depending on twelve_hour flag.
    """
    if twelve_hour:
        if time_struct.tm_hour > 12:
            hour_string = str(time_struct.tm_hour - 12) # 13-23 -> 1-11 (pm)
        elif time_struct.tm_hour > 0:
            hour_string = str(time_struct.tm_hour) # 1-12
        else:
            hour_string = '12' # 0 -> 12 (am)
    else:
        hour_string = '{hh:02d}'.format(hh=time_struct.tm_hour)
    return hour_string + ':{mm:02d}'.format(mm=time_struct.tm_min)


# GRAPHICS INITIALIZATION -------------------------------------------------

MAGTAG.graphics.set_background('bitmaps/nice.bmp' if NICE else
                               'bitmaps/naughty.bmp')

# Add empty name list here in the drawing stack, names are added later
MAGTAG.add_text(
    text_font='/fonts/cursive-smart.pcf',
    text_position=(8, 40),
    line_spacing=1.0,
    text_anchor_point=(0, 0), # Top left
    is_data=False,            # Text will be set manually
)

# Add 14-pixel-tall black bar at bottom of display. It's a distinct layer
# (not just background) to appear on top of name list if it runs long.
MAGTAG.graphics.splash.append(Rect(0, MAGTAG.graphics.display.height - 14,
                                   MAGTAG.graphics.display.width,
                                   MAGTAG.graphics.display.height, fill=0x0))

# Center white text label over black bar to show last update time
# (Initially a placeholder, string is not assigned to label until later)
MAGTAG.add_text(
    text_font='/fonts/helvB12.pcf',
    text_position=(MAGTAG.graphics.display.width // 2,
                   MAGTAG.graphics.display.height - 1),
    text_color=0xFFFFFF,
```

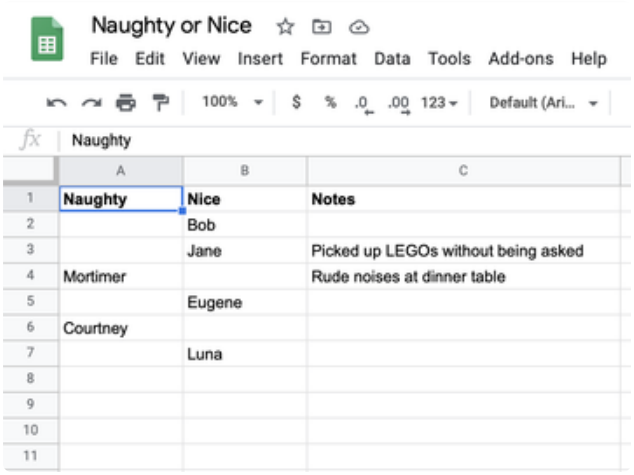
```
    text_anchor_point=(0.5, 1), # Center bottom
    is_data=False,              # Text will be set manually
)


# MAIN LOOP -----------------------------------------------------------------

# FYI: Not really a "loop" -- deep sleep makes the whole system restart on
# wake, this only needs to run once.

try:
    MAGTAG.network.connect() # Do this last, as WiFi uses power

    print('Updating time')
    MAGTAG.get_local_time()
    NOW = rtc.RTC().datetime
    print(NOW)

    print('Updating names')
    RESPONSE = MAGTAG.network.fetch(TSV_URL)
    if RESPONSE.status_code == 200:
        TSV_DATA = RESPONSE.text
        print('OK')

    # Set the "Updated" date and time label
    if DD_MM:
        DATE = '%d/%d' % (NOW.tm_mday, NOW.tm_mon)
    else:
        DATE = '%d/%d' % (NOW.tm_mon, NOW.tm_mday)
    MAGTAG.set_text('Updated %s %s' % (DATE, hh_mm(NOW, TWELVE_HOUR)), 1,
                    auto_refresh=False)

    # Split text response into separate lines
    LINES = TSV_DATA.split('\r\n')

    # Scan cells in row #1 to find the column number for naughty vs nice.
    # This allows the order of columns in the spreadsheet to be changed,
    # though they still must have a "Naughty" or "Nice" heading at top.
    cells = LINES[0].split("\t") # Tab-separated values
    for column, entry in enumerate(cells):
        head = entry.lower() # Case-insensitive compare
        if ((NICE and head == 'nice') or (not NICE and head == 'naughty')):
            NAME_COLUMN = column

    # Now that we know which column number contains the names we want,
    # a second pass is made through all the cells. Items where row > 1
    # and column is equal to NAME_COLUMN are joined in a string.
    NAME_LIST = '' # Clear name list
    for line in LINES[1:]: # Skip first line -- naughty/nice/notes in sheet
        cells = line.split("\t") # Tab-separated
        if len(cells) >= NAME_COLUMN and cells[NAME_COLUMN] != "":
            NAME_LIST += cells[NAME_COLUMN] + '\n' # Name + newline character

    MAGTAG.set_text(NAME_LIST) # Update list on the display

    time.sleep(2) # Allow refresh to finish before deep sleep
    print('Zzzz time')
    MAGTAG.exit_and_deep_sleep(24 * 60 * 60) # 24 hour deep sleep

except RuntimeError as error:
    # If there's an error above, no harm, just try again in ~15 minutes.
    # Usually it's a common network issue or time server hiccup.
    print('Retrying in 15 min - ', error)
    MAGTAG.exit_and_deep_sleep(15 * 60) # 15 minute deep sleep
```

Skateboard Tricks

A couple of things make this program more flexible. First, near the top of the code is this line:

```
NICE = True # Use 'True' for nice list, 'False' for naughty
```

This can be set to **True** or **False** depending on which list you want to see. The **same program** then gets installed on both MagTag boards...they don't each require unique code.



Second...let's take a look at that final spreadsheet layout again. Notice that each column has a **heading** (row #1), either "Naughty," "Nice" or "Notes."

Santa and Krampus are always butting heads over who's more important. Sometimes, when the other isn't looking, they'll rearrange the columns to put their job first.

By examining those headings, the program's output will always match the value of the **NICE** variable's **True** or **False** setting, even if the columns get reordered!

This sort of thing is not required. If you know for certain that Naughty and Nice will always be in a fixed order, the code could be simpler. It's just to show how things can be made more foolproof with just a few added lines.

Here's the relevant part of the code, where you can see two passes being made through the **LINES** list. The first pass looks only at the column headings (row #1 in the spreadsheet, or LINES[0] in CircuitPython) for "nice" or "naughty" (using a case-insensitive compare, again for foolproofishness) that corresponds to the global **NICE** setting. Second pass collects all the names in that column who's row number is greater than 1 (so the headings won't appear in the list).

```
# Split text response into separate lines
LINES = TSV_DATA.split('\r\n')
```

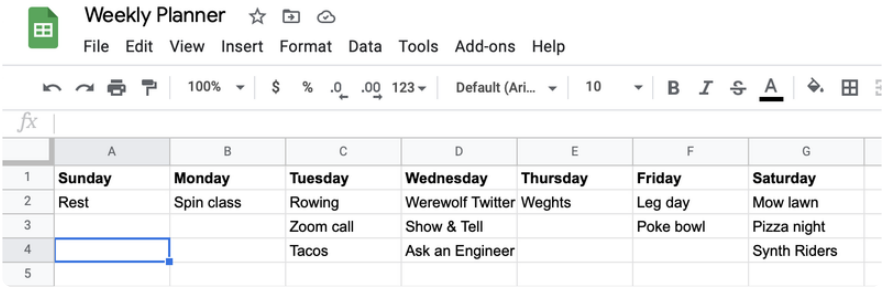
```
# Scan cells in row #1 to find the column number for naughty vs nice.
# This allows the order of columns in the spreadsheet to be changed,
# though they still must have a "Naughty" or "Nice" heading at top.
cells = LINES[0].split("\t") # Tab-separated values
for column, entry in enumerate(cells):
    head = entry.lower() # Case-insensitive compare
    if ((NICE and head == 'nice') or (not NICE and head == 'naughty')):
        NAME_COLUMN = column

# Now that we know which column number contains the names we want,
# a second pass is made through all the cells. Items where row &gt; 1
# and column is equal to NAME_COLUMN are joined in a string.
NAME_LIST = '' # Clear name list
for line in LINES[1:]: # Skip first line -- naughty/nice/notes in sheet
    cells = line.split("\t") # Tab-separated
    if len(cells) &gt;= NAME_COLUMN and cells[NAME_COLUMN] != "":
        NAME_LIST += cells[NAME_COLUMN] + '\n' # Name + newline character
```

To avoid blank lines in the displayed name list, the code above intentionally checks for empty cells (`""`) and ignores them.

Example: Weekly Planner

Let's consider a more practical example: a planner that shows your recurring to-dos for each day of the week.

The spreadsheet might look like the following, with one column for each day of the week, and a varying number of tasks down the rows for each day. The first row contains day names:



	A	B	C	D	E	F	G
1	**Sunday**	**Monday**	**Tuesday**	**Wednesday**	**Thursday**	**Friday**	**Saturday**
2	Rest	Spin class	Rowing	Werewolf Twitter	Weghts	Leg day	Mow lawn
3			Zoom call	Show & Tell		Poke bowl	Pizza night
4			Tacos	Ask an Engineer			Synth Riders
5							

(European calendars usually show Monday as the first day of the week. That's entirely possible here, with a corresponding change in the code...this is covered later.)

This has some **things in common** with the "Naughty or Nice" example:

- We'll display only the information from **one column**
- The headers (row #1) are **ignored**

And some **differences**:

- The columns are always going to be in this order, so there's no need for a first pass to read the headings...just use the column number (1–7) corresponding to the current day of the week (Sunday–Saturday)
- The **column selection** is based on the **system clock** (polled from an internet time server), not a global variable in the code

Make and Publish the Spreadsheet

With a Google Sheet resembling the above, follow the directions on the “Spreadsheet Setup” page to get the data ready for MagTag. Publish to web, copy the document’s URL, and then paste this link into the code. You only need to do this part once, any document changes will use the same link.

CircuitPython Code

Here’s the CircuitPython code for the weekly planner (**weekly_planner.py**). Once again, the spreadsheet layout and our CircuitPython code are designed in concert. The vital bits were already explained on the “Parsing” page, and much of it is recycled from the prior example.

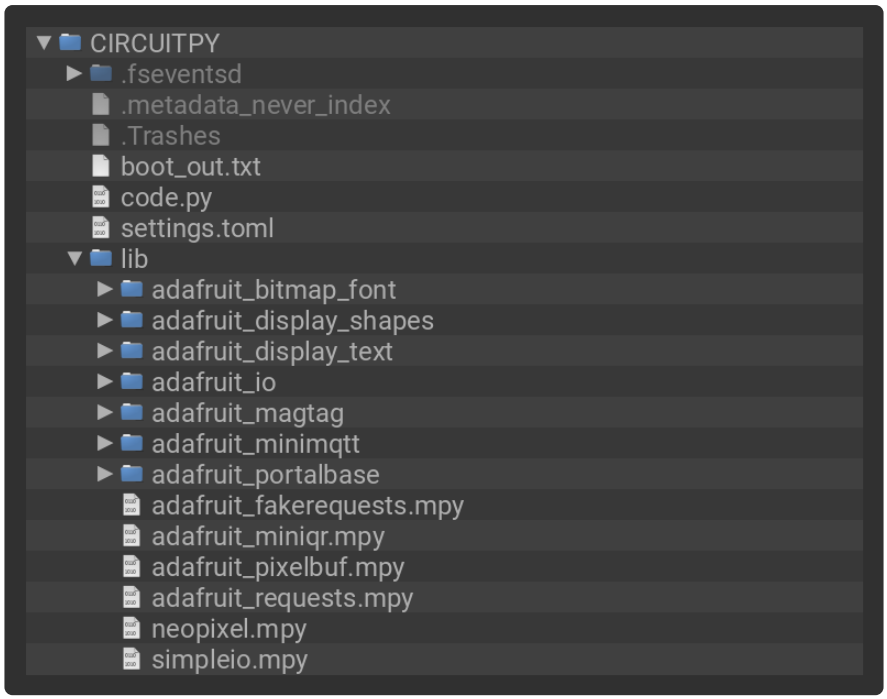
Update TSV_URL to point to your published spreadsheet, and rename the file to “code.py” to run automatically on the CIRCUITPY drive.

Installing Project Code

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

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

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



If you're having difficulty running this example, it could be because your MagTag CircuitPython firmware or library needs to be upgraded! Please be sure to follow https://learn.adafruit.com/adafruit-magtag/circuitpython to install the latest CircuitPython firmware and then also replace/update ALL the MagTag-specific libraries mentioned here https://learn.adafruit.com/adafruit-magtag/circuitpython-libraries-2

```
# SPDX-FileCopyrightText: 2020 Phillip Burgess for Adafruit Industries
#
# SPDX-License-Identifier: MIT

"""
Google Sheets to MagTag example: Weekly Planner.
Gets tab-separated-value (TSV) spreadsheet from Google, displays task list
from today's column. This example does NOT deep sleep, a USB power connection
is recommended.
Fonts from Xorg project.
"""

# pylint: disable=import-error, line-too-long
import time
import rtc
from adafruit_display_shapes.rect import Rect
from adafruit_magtag.magtag import MagTag


# CONFIGURABLE SETTINGS and ONE-TIME INITIALIZATION ----------------------

TSV_URL = 'https://docs.google.com/spreadsheets/d/e/2PACX-1vR1WjUKz35-
ek6SiR5droDfvPp51MTds4wUs57vEZNh2uDfihSTPhTaiiRovLbNe1mkeRgurppRJ_Zy/pub?output=tsv'
TWELVE_HOUR = True # If set, show 12-hour vs 24-hour (e.g. 3:00 vs 15:00)
DD_MM = False      # If set, show DD/MM instead of MM/DD dates
DAYS = ['Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday',
        'Saturday']

MAGTAG = MagTag(rotation=0) # Portrait (vertical) display
MAGTAG.network.connect()
```

```
# SOME UTILITY FUNCTIONS ------------------------------------------------

def hh_mm(time_struct, twelve_hour=True):
    """ Given a time.struct_time, return a string as H:MM or HH:MM, either
        12- or 24-hour style depending on twelve_hour flag.
    """
    if twelve_hour:
        if time_struct.tm_hour > 12:
            hour_string = str(time_struct.tm_hour - 12) # 13-23 -> 1-11 (pm)
        elif time_struct.tm_hour > 0:
            hour_string = str(time_struct.tm_hour) # 1-12
        else:
            hour_string = '12' # 0 -> 12 (am)
    else:
        hour_string = '{hh:02d}'.format(hh=time_struct.tm_hour)
    return hour_string + ':{mm:02d}'.format(mm=time_struct.tm_min)


# GRAPHICS INITIALIZATION -----------------------------------------------

# First text label (index 0) is day of week -- empty for now, is set later
MAGTAG.add_text(
    text_font='/fonts/helvB24.pcf',
    text_position=(MAGTAG.graphics.display.width // 2, 4),
    line_spacing=1.0,
    text_anchor_point=(0.5, 0), # Center top
    is_data=False,              # Text will be set manually
)

# Second (index 1) is task list -- again, empty on start, is set later
MAGTAG.add_text(
    text_font='/fonts/ncenR14.pcf',
    text_position=(3, 36),
    line_spacing=1.0,
    text_anchor_point=(0, 0), # Top left
    is_data=False,            # Text will be set manually
)

# Add 14-pixel-tall black bar at bottom of display. It's a distinct layer
# (not just background) to appear on top of task list if it runs long.
MAGTAG.graphics.splash.append(Rect(0, MAGTAG.graphics.display.height - 14,
                                   MAGTAG.graphics.display.width,
                                   MAGTAG.graphics.display.height, fill=0x0))

# Center white text (index 2) over black bar to show last update time
MAGTAG.add_text(
    text_font='/fonts/helvB12.pcf',
    text_position=(MAGTAG.graphics.display.width // 2,
                   MAGTAG.graphics.display.height - 1),
    text_color=0xFFFFFF,
    text_anchor_point=(0.5, 1), # Center bottom
    is_data=False,              # Text will be set manually
)


# MAIN LOOP -------------------------------------------------------------

PRIOR_LIST = '' # Initialize these to nonsense values
PRIOR_DAY = -1  # so the list or day change always triggers on first pass

while True:
    try:
        print('Updating time')
        MAGTAG.get_local_time()
        NOW = rtc.RTC().datetime

        print('Updating tasks')
```

```
        RESPONSE = MAGTAG.network.fetch(TSV_URL)
        if RESPONSE.status_code == 200:
            TSV_DATA = RESPONSE.text
            print('OK')

        # Split text response into separate lines
        LINES = TSV_DATA.split('\r\n')

        # tm_wday uses 0-6 for Mon-Sun, we want 1-7 for Sun-Sat
        COLUMN = (NOW.tm_wday + 1) % 7 + 1

        TASK_LIST = '' # Clear task list string
        for line in LINES[1:]: # Skip first line -- days of week in sheet
            cells = line.split("\t") # Tab-separated!
            if len(cells) >= COLUMN:
                TASK_LIST += cells[COLUMN - 1] + '\n'

        # Refreshing the display is jarring, so only do it if the task list
        # or day has changed. This requires preserving state between passes,
        # and is why this code doesn't deep sleep (which is like a reset).
        if TASK_LIST != PRIOR_LIST or PRIOR_DAY != NOW.tm_wday:

            # Set the day-of-week label at top
            MAGTAG.set_text(DAYS[COLUMN - 1], auto_refresh=False)

            # Set the "Updated" date and time label
            if DD_MM:
                DATE = '%d/%d' % (NOW.tm_mday, NOW.tm_mon)
            else:
                DATE = '%d/%d' % (NOW.tm_mon, NOW.tm_mday)
            MAGTAG.set_text('Updated %s %s' % (DATE, hh_mm(NOW, TWELVE_HOUR)),
                            2, auto_refresh=False)

            MAGTAG.set_text(TASK_LIST, 1) # Update list, refresh display
            PRIOR_LIST = TASK_LIST        # Save list state for next pass
            PRIOR_DAY = NOW.tm_wday       # Save day-of-week for next pass

    except RuntimeError as error:
        # If there's an error above, no harm, just try again in ~15 minutes.
        # Usually it's a common network issue or time server hiccup.
        print('Retrying in 15 min - ', error)

    time.sleep(15 * 60) # Whether OK or error, wait 15 mins for next pass
```

Skateboard Tricks

Unlike the previous example which uses deep sleep (waking once a day), this one requires **continuous USB power** because it's **frequently checking the spreadsheet for changes**, about four times an hour...a battery wouldn't last a day with that much WiFi access going on. Because e-ink screen updates can be rather jarring, the code only refreshes the display if it **detects a change** in the list or the day of the week. The "list" is really just a long string, maybe a couple hundred bytes tops, so it's not unreasonable to keep that in memory, and is easy to compare.

Earlier it was mentioned how European calendars start their week on Monday. Time functions in Python also start the week on Monday, but because this code is US-centric, there's a weird line of code to map from Python day-of-week (starting at 0 for Monday) to a spreadsheet column number (starting at 1 for Sunday):

```
DAYS = ['Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday',
'Saturday']

# tm_wday uses 0-6 for Mon-Sun, we want 1-7 for Sun-Sat
COLUMN = (NOW.tm_wday + 1) % 7 + 1

# Set the day-of-week label at top
MAGTAG.set_text(DAYS[COLUMN - 1], auto_refresh=False)
```

It looks odd, adding 1 to COLUMN only to subtract 1 on the next line, but that's because this operation is only used once, whereas the subsequent scan for cells matching COLUMN makes this comparison many times over, so we optimize for that case.

For a Euro calendar layout (with matching spreadsheet), this could be:

```
DAYS = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday',
'Sunday']

COLUMN = NOW.tm_wday + 1

MAGTAG.set_text(DAYS[NOW.tm_wday], auto_refresh=False)
```

Strictly speaking, it would be "more efficient" to just always use the Euro format. But from an overall design standpoint, I'd say "better" to use a layout that the end user is more familiar with, to avoid entering things in the wrong column. A more complete version of this code might use a variable to select between the two formats, but then this starts to turn into a rant about user-centered design when the goal here is just a lightweight introduction Google Sheets and CircuitPython.

The example code uses a fixed column number for each day rather than scanning the headers for a match (as in the prior example). Partly for simplicity, but also because "Wednesday" is often misspelled and might not be found in a search through the headers.