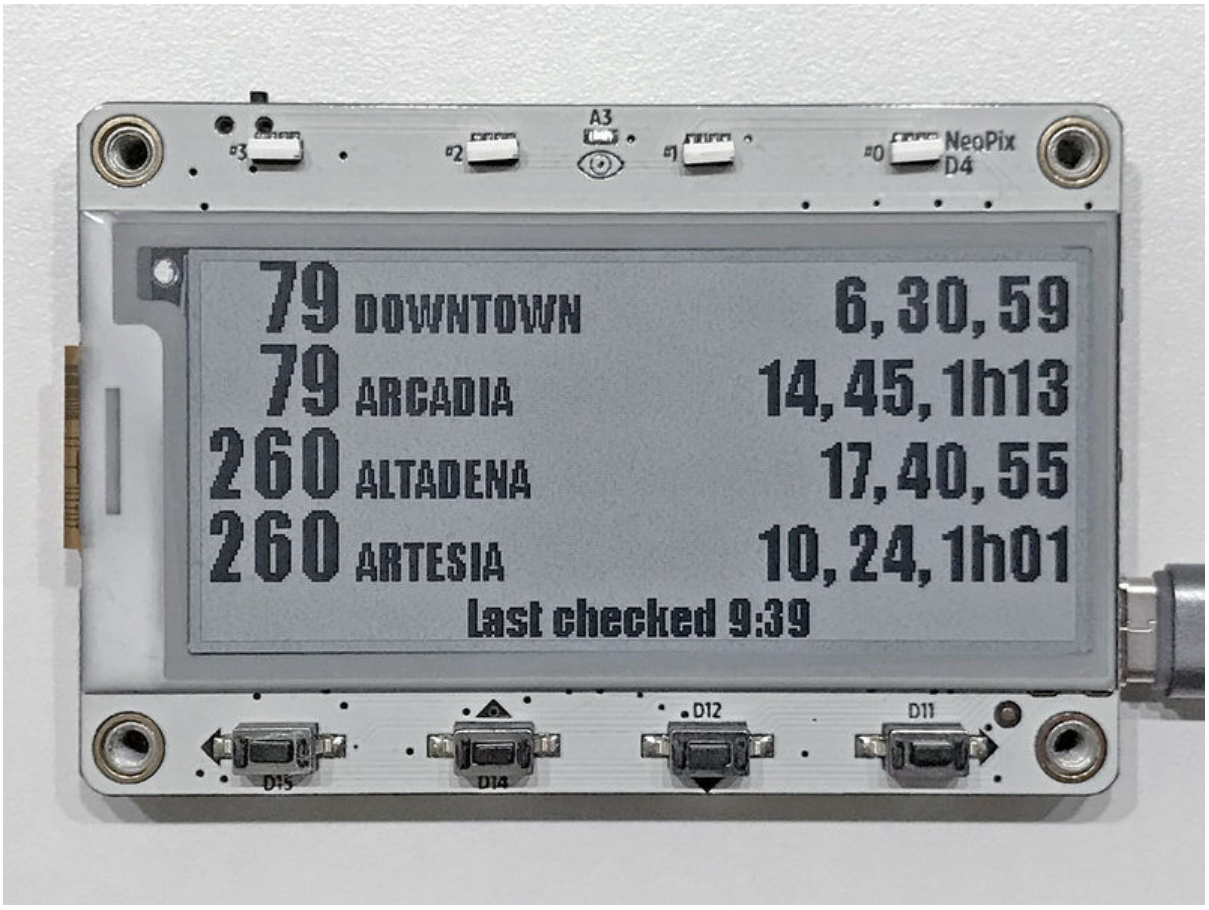
NextBus Transit Predictions for Adafruit MagTag

Created by Phillip Burgess

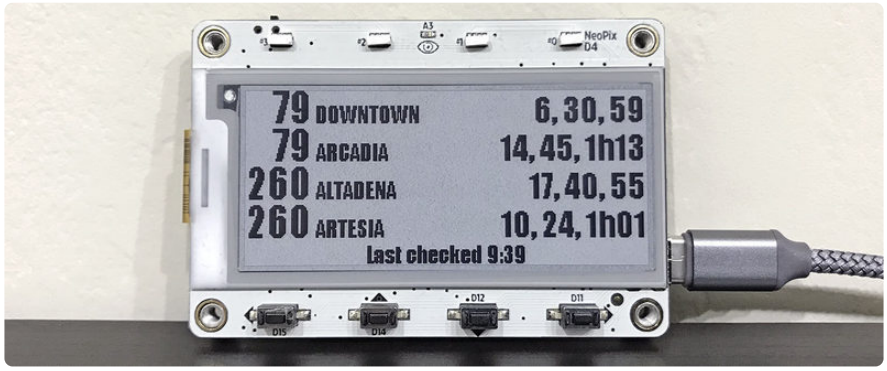


https://learn.adafruit.com/nextbus-transit-predictions-for-adafruit-magtag

Last updated on 2024-11-29 10:05:37 AM EST

Table of Contents

Overview



NextBus **(https://adafru.it/eCA)** is a free internet service using GPS and cellular networks to provide realtime arrival data for 60+ transit agencies in the United States and Canada.

For transit-bound people, the NextBus service is a tremendous convenience. Knowing when a bus is due means less standing out in the rain...one can use that time inside to get a little extra work done, or finish that cup of coffee.

NextBus provides web and mobile phone access, and there are some nice smartphone apps around. As a “heavy user,” I wanted to take it one step further, creating a wall clock of sorts...a continuous feed of up to four stops/routes relevant to my needs...no need to even pull out a phone or click a bookmark, the information’s always there at a glance.

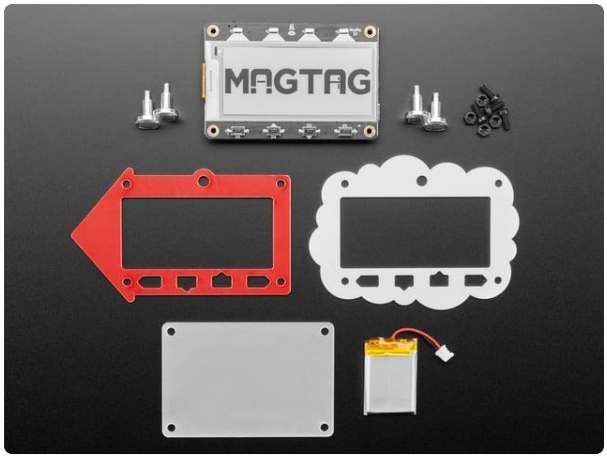
Parts and Tools Required

- Adafruit MagTag **(http://adafru.it/4800)** E-Ink WiFi Display
- **WiFi network** (802.11 b/g/n)
- **USB-C cable** and **power source**
- A desktop or laptop computer is required for initial setup: any text editor will suffice, and a **Python 3** interpreter

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...

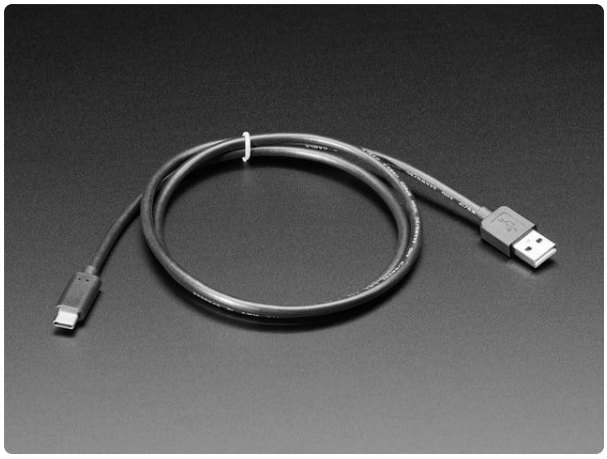
https://www.adafruit.com/product/4800



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



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

Setup

Plug It In

Unlike the majority of MagTag projects that can run for weeks on a single battery charge, this project relies on **frequent WiFi access** and is best handled with a **continuous USB power source**. You'll need a **USB-C cable** and a nearby USB hub or a small phone charger (most folks have accumulated several spares by now). The project could be made standalone, but you'd need a sizable LiPoly battery if it's to run for an appreciable length of time.

Snakes on a Plane

While the NextBus clock code runs self-contained on the MagTag using CircuitPython, **initial setup requires access to "full" Python 3** on a regular computer, and a bit of command-line typing in a terminal window.

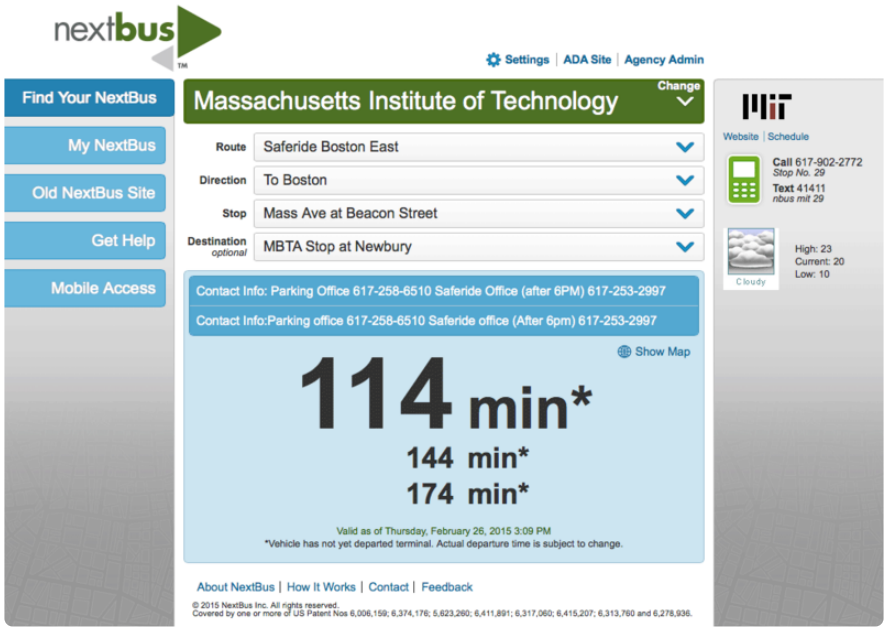
Some systems (e.g. Raspberry Pi) already have Python 3 installed. Others, like Windows and Mac, may require an install. If unsure, open a terminal window and type "**python3**" — if you get an error, installation is required. **Visit the** Python.org download page **(https://adafru.it/fa7) for guidance.**

Realistic Expectations

Before commiting to this project, I'd suggest trying the NextBus service for a couple weeks with your regular web browser and/or on your phone, in order to understand its limitations.

While very convenient and fairly reliable overall, the system is not 100% perfect. Not all vehicles are equipped with working tracking hardware. Occasionally GPS or cell signals are lost and tracking estimates may jump forward or back by several minutes.

Get to know how much lead time you need to safely and reliably make your transit connection, and whether the service meets your needs. I find it most useful for deciding whether to run errands now versus later.



If you're still on board, let's get started...

Install CircuitPython

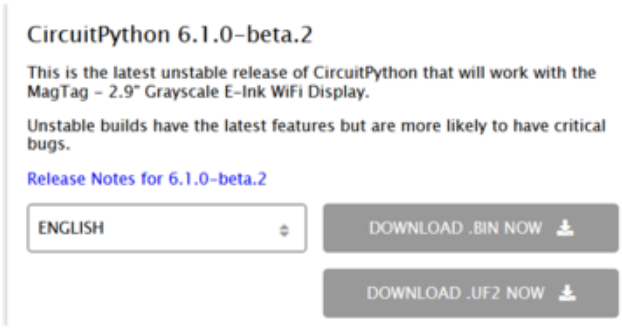
CircuitPython (https://adafru.it/tB7) is a derivative of MicroPython (https://adafru.it/BeZ) designed to simplify experimentation and education on low-cost microcontrollers. It makes it easier than ever to get prototyping by requiring no upfront desktop software downloads. Simply copy and edit files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

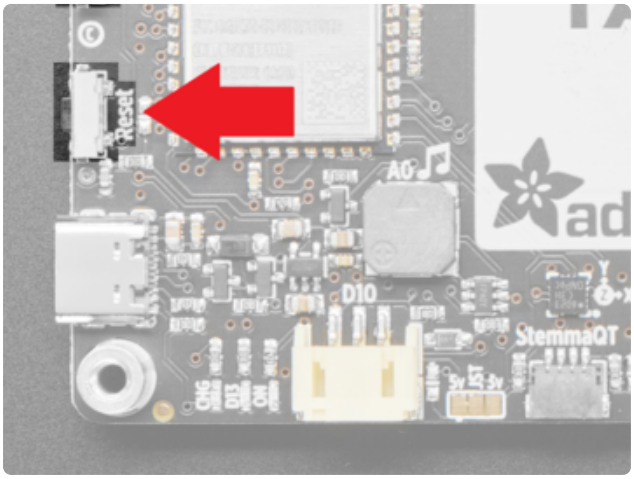
Option 1 - Load with UF2 Bootloader

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.**

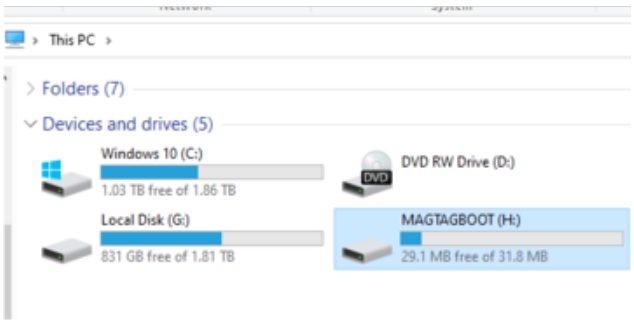
Still, try this first!

Try Launching UF2 Bootloader

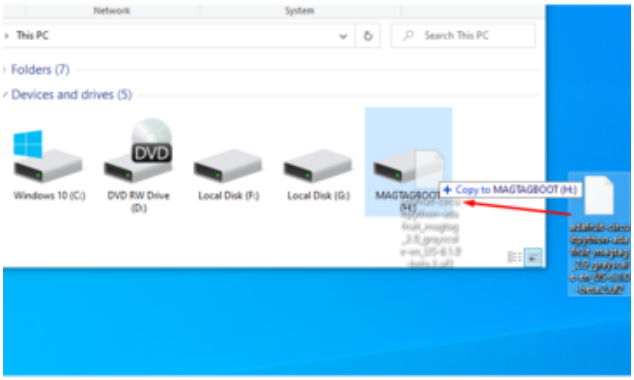
Loading CircuitPython by drag-n-drop UF2 bootloader is the easier way and we recommend it. If you have a MagTag where the front of the board is black, your MagTag came with UF2 already on it.



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.

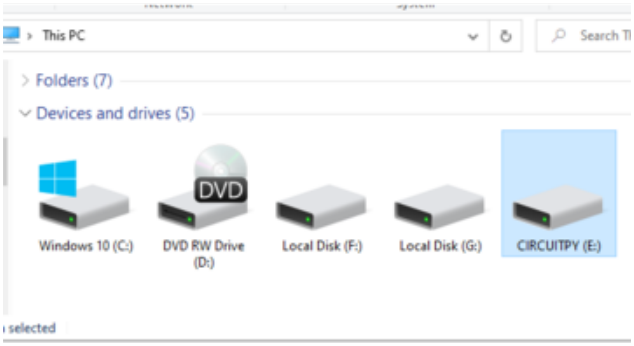


If the UF2 bootloader is installed, you will see a new disk drive appear called **MAGTAGBOOT**



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)

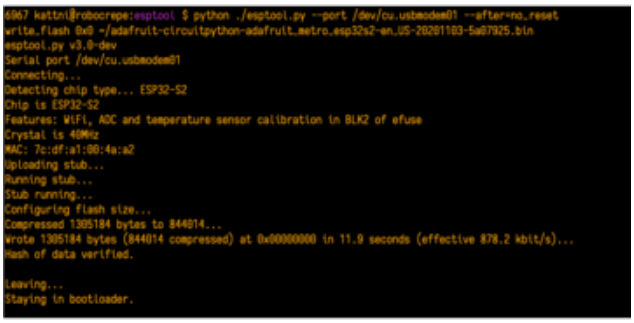


Your board should auto-reset into CircuitPython, or you may need to press reset. A **CIRCUITPY** drive will appear. You're done! Go to the next pages.

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.

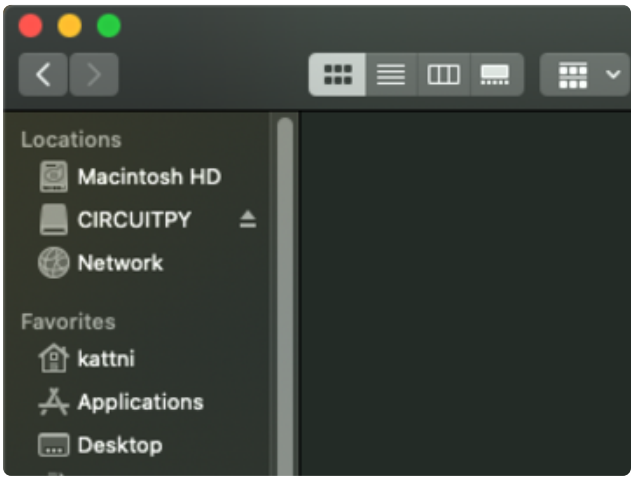
You can upload with **esptool** to the ROM (hardware) bootloader instead!



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!

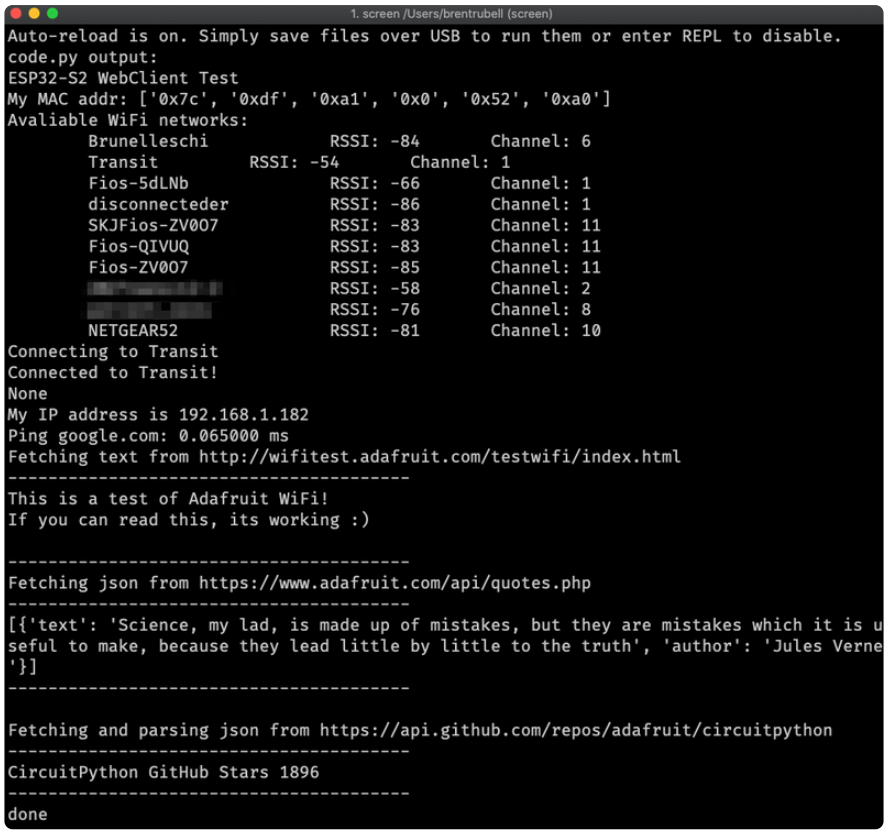
As you make projects you may need more tokens and keys, just add them one line at a time. See for example other tokens such as one for accessing GitHub or the Hackaday API. Other non-secret data like your timezone can also go here.

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!

IPv6 Networking

Starting in CircuitPython 9.2, IPv6 networking is available on most Espressif wifi boards. Socket-using libraries like **adafruit_requests** and **adafruit_ntp** will need to be updated to use the new APIs and for now can only connect to services on IPv4.

IPv6 connectivity & privacy

IPv6 addresses are divided into many special kinds, and many of those kinds (like those starting with **FC**, **FD**, **FE**) are private or local; Addresses starting with other prefixes like **2002:** and **2001:** are globally routable. In 2024, far from all ISPs and home networks support IPv6 internet connectivity. For more info consult resources like Wikipedia (https://adafru.it/1a4z). If you're interested in global IPv6 connectivity you can use services like Hurricane Electric (https://adafru.it/1a4A) to create an "IPv6 tunnel" (free as of 2024, but requires expertise and a compatible router or host computer to set up)

It's also important to be aware that, as currently implemented by Espressif, there are privacy concerns especially when these devices operate on the global IPv6 network: The device's unique identifier (its EUI-64 or MAC address) is used by default as part of its IPv6 address. This means that the device identity can be tracked across multiple networks by any service it connects to.

Enable IPv6 networking

Due to the privacy consideration, IPv6 networking is not automatically enabled. Instead, it must be explicitly enabled by a call to `start_dhcp_client` with the `ipv6=True` argument specified:

```
wifi.start_dhcp_client(ipv6=True)
```

Check IP addresses

The read-only `addresses` property of the `wifi.radio` object holds all addresses, including IPv4 and IPv6 addresses:

```
&gt;&gt;&gt; wifi.radio.addresses
('FE80::7EDF:A1FF:FE00:518C', 'FD5F:3F5C:FE50:0:7EDF:A1FF:FE00:518C', '10.0.3.96')
```

The `wifi.radio.dns` servers can be IPv4 or IPv6:

```
&gt;&gt;&gt; wifi.radio.dns
('FD5F:3F5C:FE50::1',)
```

```
&gt;&gt;&gt; wifi.radio.dns = ("1.1.1.1",)
&gt;&gt;&gt; wifi.radio.dns
('1.1.1.1',)
```

Ping v6 networks

`wifi.radio.ping` accepts v6 addresses and names:

```
&gt;&gt;&gt; wifi.radio.ping("google.com")
0.043
&gt;&gt;&gt; wifi.radio.ping("ipv6.google.com")
0.048
```

Create & use IPv6 sockets

Use the address family `socket.AF_INET6`. After the socket is created, use methods like `connect`, `send`, `recfrom_into`, etc just like for IPv4 sockets. This code snippet shows communicating with a private-network NTP server; this IPv6 address will not work on your network:

```
&gt;&gt;&gt; ntp_addr = ("fd5f:3f5c:fe50::20e", 123)
&gt;&gt;&gt; PACKET_SIZE = 48
&gt;&gt;&gt;
&gt;&gt;&gt; buf = bytearray(PACKET_SIZE)
&gt;&gt;&gt; with socket.socket(socket.AF_INET6, socket.SOCK_DGRAM) as s:
...     s.settimeout(1)
...     buf[0] = 0b0010_0011
...     s.sendto(buf, ntp_addr)
...     print(s.recvfrom_into(buf))
...     print(buf)
...
48
(48, ('fd5f:3f5c:fe50::20e', 123))
bytearray(b'$\x01\x03\xeb\x00\x00\x00\x00\x00\x00\x00GGPS\x00\xeaA0h\x07s;
\xc0\x00\x00\x00\x00\x00\x00\x00\xeaA0n\xeb4\x82-\xeaA0n\xebAU\xb1')
```

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

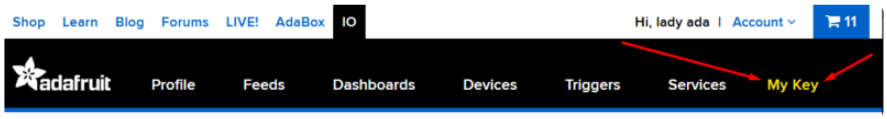
It's free! Visit https://accounts.adafruit.com/ (https://adafru.it/dyy) to register and make an account if you do not already have one

Step 2) Sign into Adafruit IO

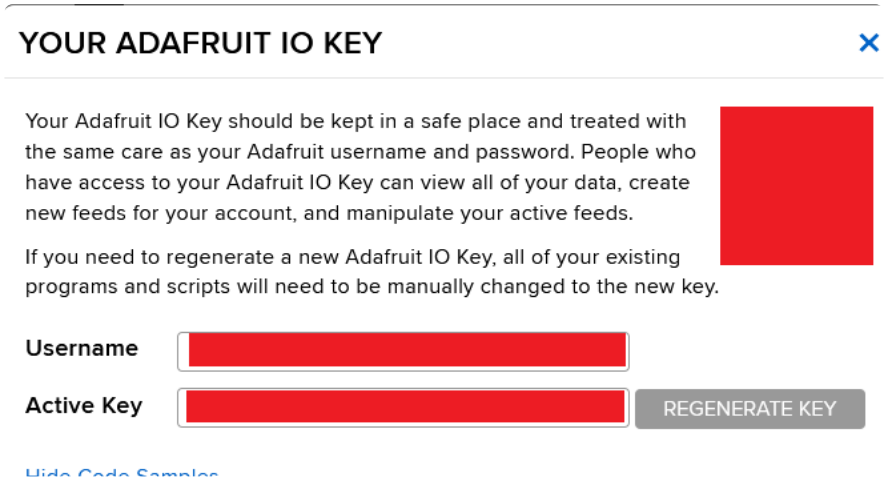
Head over to io.adafruit.com (https://adafru.it/fsU) and click **Sign In** to log into IO using your Adafruit account. It's free and fast to join.

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_USERNAME = "your-adafruit-io-username"
ADAFRUIT_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).

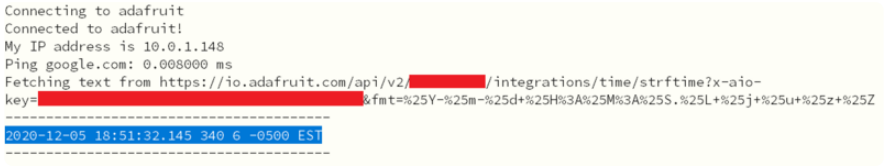
Step 4) Upload Test Python Code

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!

MagTag-Specific CircuitPython Libraries

To use all the amazing features of your MagTag with CircuitPython, you must first install a number of libraries. This page covers that process.

Get Latest Adafruit CircuitPython Bundle

Download the Adafruit CircuitPython Library Bundle. You can find the latest release here:

Download the latest Library Bundle from circuitpython.org

https://adafru.it/ENC

Download the **adafruit-circuitpython-bundle-version-mpy-*.zip** bundle zip file, and unzip a folder of the same name. Inside you'll find a **lib** folder. The entire collection of libraries is too large to fit on the **CIRCUITPY** drive. Therefore, you'll need to copy the necessary libraries to your board individually.

At a minimum, the following libraries are required. Copy the following folders or .mpy files to the **lib** folder on your **CIRCUITPY** drive. **If the library is a folder, copy the entire folder** to the **lib** folder on your board.

Library folders (copy the whole folder over to lib):

- **adafruit_magtag** - This is a helper library designed for using all of the features of the MagTag, including networking, buttons, NeoPixels, etc.
- **adafruit_portalbase** - This library is the base library that adafruit_magtag is built on top of.

- **adafruit_bitmap_font** - There is fancy font support, and it's easy to make new fonts. This library reads and parses font files.
- **adafruit_display_text** - This library displays text on the screen.
- **adafruit_io** - This library helps connect the MagTag to our free data logging and viewing service
- **adafruit_minimqtt** - This library provides MQTT service for Adafruit IO.

Library files:

- **adafruit_requests.mpy** - This library allows us to perform HTTP requests and get responses back from servers. GET/POST/PUT/PATCH - they're all in here!
- **adafruit_fakerequests.mpy** - This library allows you to create fake HTTP requests by using local files.
- **adafruit_miniqr.mpy** - QR creation library lets us add easy-to-scan 2D barcodes to the E-Ink display
- **neopixel.mpy** - This library is used to control the onboard NeoPixels.
- **simpleio.mpy** - This library is used for tone generation.

Secrets

Even if you aren't planning to go online with your MagTag, you'll need to have a **secrets.py** file in the root directory (top level) of your **CIRCUITPY** drive. If you do not intend to connect to wireless, it does not need to have valid data in it. Here's more info on the secrets.py file (https://adafru.it/P3b).

Install Code and Graphics

Back up any existing code or files you want to keep from your MagTag CIRCUITPY drive.

Fetch the files for our NextBus clock from Github:

Download MagTag NextBus Project

https://adafru.it/19F4

Unzip this file after downloading.

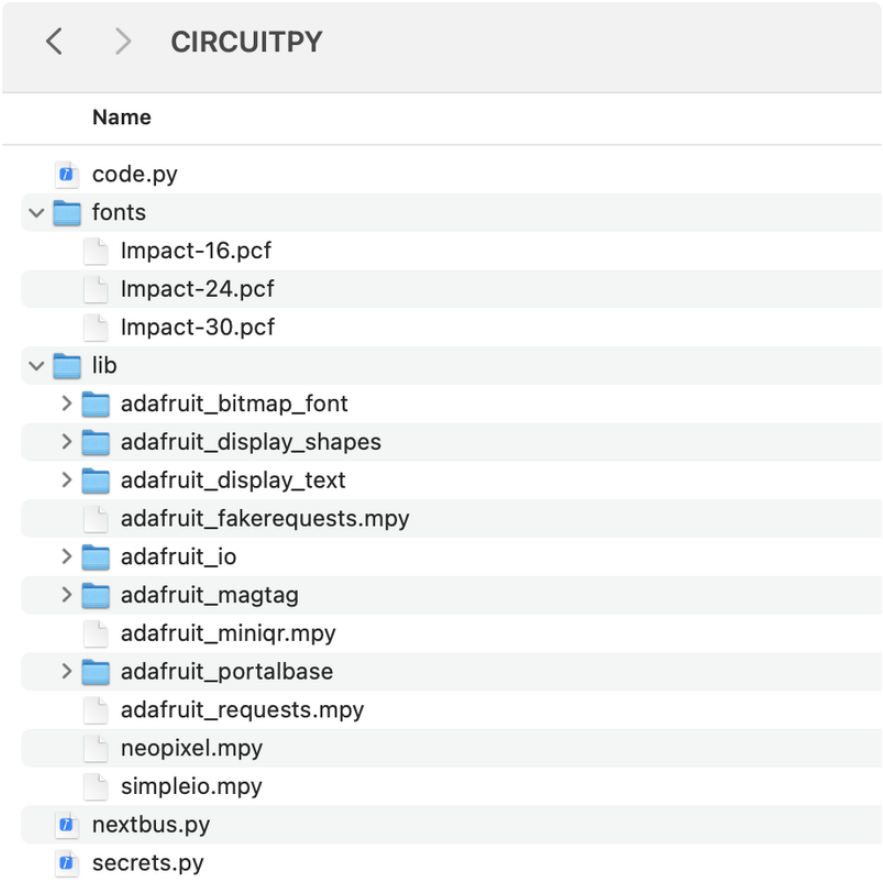
The **bitmaps** and **fonts** folders should be copied into the **CIRCUITPY** root directory. If folders with these names already exist, copy the individual .BMP and .BDF files into the corresponding folders.

code.py and **nextbus.py** should also be copied to the **CIRCUITPY** root directory.

One of the files in the project folder — **nextbus_routefinder.py** — **does not get copied**...that's "full" Python code which we'll use on the next page.

One additional file — **secrets.py** — isn't distributed in the project folder...if you don't already have this file from a prior MagTag project, we'll create this file on the next page.

Here's a map of all this project's required images, fonts and code on the **CIRCUITPY** drive:



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.

Configure Network and Transit Selections

secrets.py holds your **WiFi network credentials** and other info. This file can be created or edited with any simple text editor you prefer.

If you already have this file on your MagTag from prior projects...great!

If not, it should resemble what's below, with the **ssid** and **password** lines, edited to match your WiFi network credentials. The corresponding values for these (after the colon) are in single-quotes. Other projects may add their own special lines.

The format of this file is super persnickety, every space and comma counts! If creating it for the first time, best to copy-and-paste the text below exactly, then change any items of interest (preserving quotation marks and such).

```
secrets = {
    'ssid'     : 'WiFi-Network-Name',
    'password' : 'WiFi-Network-Password'
}
```

The code will relaunch any time there's a change on the **CIRCUITPY** drive...so, after editing **secrets.py**, the clock should start up on its own and within a minute or so you'll see some initial bus predictions (we'll configure the routes and stops in a moment).

If the clock does NOT start up: most likely the WiFi credentials are incorrect, or something is wrong with the **secrets.py** file syntax...make sure every quote, comma and colon is there and in the right place.

Configuring Stops and Routes

The clock can display predictions for **up to four** transit stop/route combos. So...for example...there are two principal bus lines that run nearest my house, with stops on the near and far sides of the street as the buses run in alternate directions...four permutations in total.

This is the part of the project that requires use of **Python** from the **command line**.

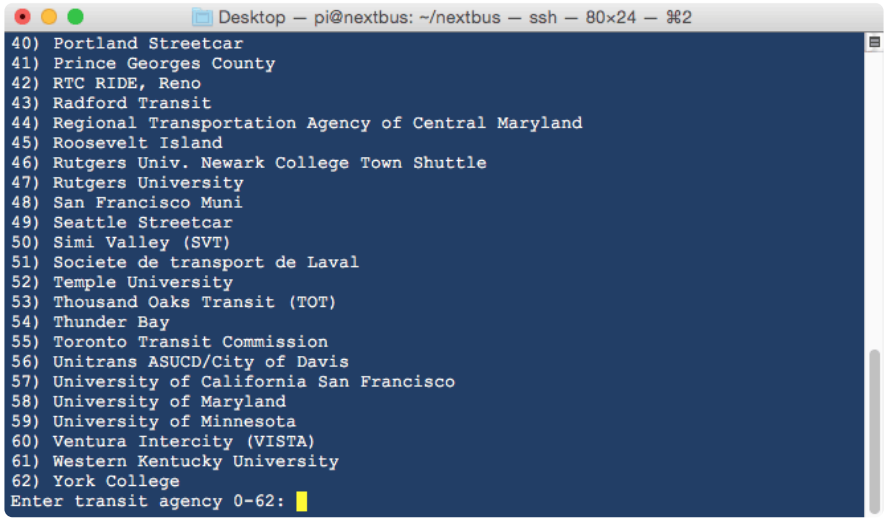
Included in the project folder is a file called **nextbus_routefinder.py**. This file does not go on the microcontroller board...you use this one on your regular computer to look up information about transit agencies and stops. Once configured, you won't need it anymore, unless you want to change the settings later.

From the command line:

```
python3 nextbus_routefinder.py
```

(You might be able to type just “python” if “python3” throws an error.)

This is an old-school text application that will guide you through a list of transit agencies, route numbers, directions and stops. Just type the number corresponding to the item of interest (sometimes the lists are long, so it helps to have a terminal program with scroll-back capability):



After making all your selections, the program then spits out a message like the following:

```
COPY/PASTE INTO APPLICATION SCRIPT:
    ('lametro', '79', '2549', 'Arcadia Station'),
```

That second line…the part in parenthesis…you’ll want to **copy and paste** that into a specific part of the **code.py** file on the **CIRCUITPY** drive…

Look for a block of code resembling the following, starting around line 28 or so:

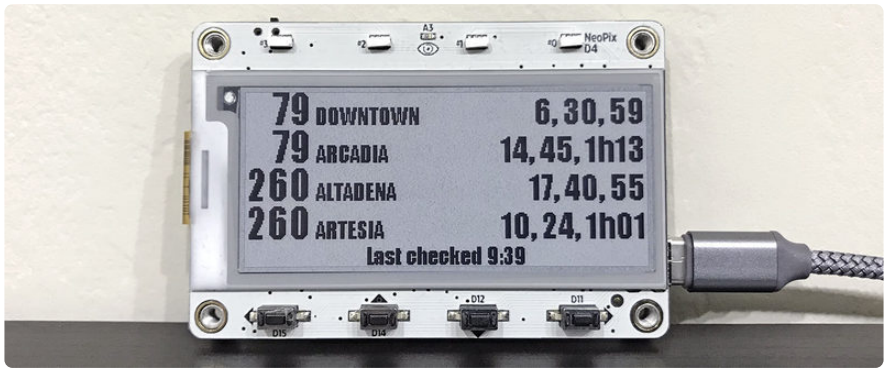
```
STOPS = [
    ('lametro', '79', '11086', 'Downtown'),
    ('lametro', '79', '2549', 'Arcadia'),
    ('lametro', '260', '11086', 'Altadena'),
    ('lametro', '260', '2549', 'Artesia')
]
```

Replace one of those lines with the line you copied from the routefinder output, then save the file.

Repeat running **nextbus_routefinder.py** for each additional route and stop you want (up to a maximum of four), copying each into code.py. Any extra lines can be deleted if using fewer than four.

One extra step you can optionally perform: space on the MagTag display is limited, so it's helpful to abbreviate each route's description. In the example above, NextBus described one route/direction as "Arcadia Station" … but, since this is the only route making any mention of Arcadia, the example code abbreviated this to "Arcadia" ("Station" was manually removed). The others were all shortened as well (e.g. "Downtown LA" is just "Downtown" in example).

Clock Usage



Once routes/stops are configured and WiFi is working…there's literally **nothing to do**. About **once a minute** the screen will update with the latest predictions (arrival times in minutes, or hours and minutes), and every few minutes it will contact the NextBus server to synchronize its predictions.

There are no buttons to press or menus to navigate. This is an intentional design decision, based on my own experience with using the service. It's most useful if the information is all simply there, now. If you're pressing buttons, you might already be missing a bus.

One of e-ink's most endearing attributes — maintaining an image even without power — is a potential liability in an application like this. If the program crashes or if the WiFi network can't be reached, the last predictions would still be on the screen, getting progressively more wrong. The "Last checked" time at the bottom of the display lets you know when the NextBus server was last contacted…this should never be more than a few minutes off track. The code does its best to recover gracefully from errors…but if something does go wrong, the "Last checked" time is how you'll know.

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

Other Configurable Settings

Aside from the transit routes and stops, there are some other configurable settings that can (and sometimes should) be adjusted.

In **code.py**, below the list of stops, are a few global variables. Each of these is pretty well commented in the code, but for posterity...

`QUERY_INTERVAL` determines how often to contact the NextBus server and update predictions, in seconds. By default this is set for four minutes (`4 * 60` seconds). NextBus does have bandwidth caps, so you don't want this too frequent.

`MAX_PREDICTIONS` limits how many predictions to show for each route. NextBus allows up to 5...but screen space is limited, and really I've never found that many predictions to be helpful. Like weather prediction, things get more uncertain the farther you look ahead. The default here is `3`, though it may show less if that's all the data available from NextBus (usually at a bus route's end-of-day).

`MINIMUM_TIME` is a limit (in seconds) below which arrivals will not be shown. By default this is 5 minutes (`5 * 60` seconds). You'll want to configure this for your reality. For example, I know that a brisk walk will get me to any of my stops in about six minutes...five if I really boogie. Any less, I just don't want to know, and I'll plan my time around the next arrival. Bus Zen.

I feel very strongly about this. Bad things can happen when you hurry...falls, injuries, crossing busy streets unsafely...seen it all. Sometimes ignorance is bliss. Set a reasonable `MINIMUM_TIME` and just catch the next bus alive!

`CLOCK_SYNC_INTERVAL` sets a time (in seconds) to sync up the MagTag's internal clock with an internet time server, because it can't quite maintain perfect time on its own. This really only affects the "Last time" shown at the bottom of the display. Prediction times will always work regardless, since these come from NextBus. Default is every 6 hours (`6 * 60 * 60` seconds).

`TIME_ZONE` shouldn't be set in the code...you add this to the same **secrets.py** file that holds your WiFi credentials, like so:

```
secrets = {
    'ssid'     : 'WiFi-Network-Name',
    'password' : 'WiFi-Network-Password',
    'timezone' : 'America/Los_Angeles'
}
```

(Note the comma added at the end of the 'password' line...every item in this list except the last should end with a comma.)

A list of valid time zone strings can be found here (https://adafru.it/OMf). These are associated with cities, not traditional timezone names, so you may need to search around to find one that matches up with your longitude.

If you don't set this up, that's okay! The time server does a pretty good job of geolocation — estimating your location and time zone based on internet address.