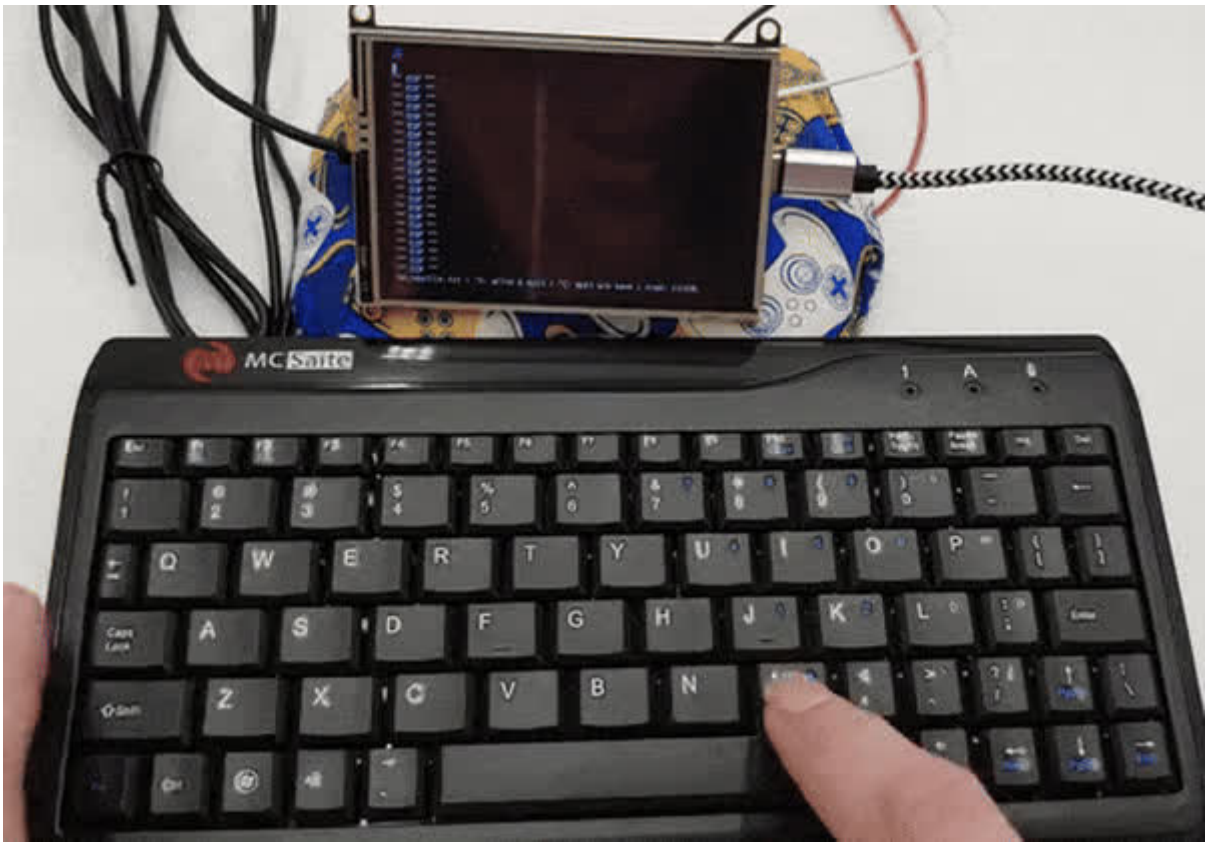
CircuitPython Text Editor On The Go

Created by Tim C



https://learn.adafruit.com/circuitpython-text-editor-on-the-go

Last updated on 2025-02-07 08:39:03 PM EST

Table of Contents

Overview

Want to make sure you can work on the Great American Novel without distraction? This project is a self-contained Text Editor that can be taken on the go when powered from a USB battery, or 2-pin JST LiPo battery.

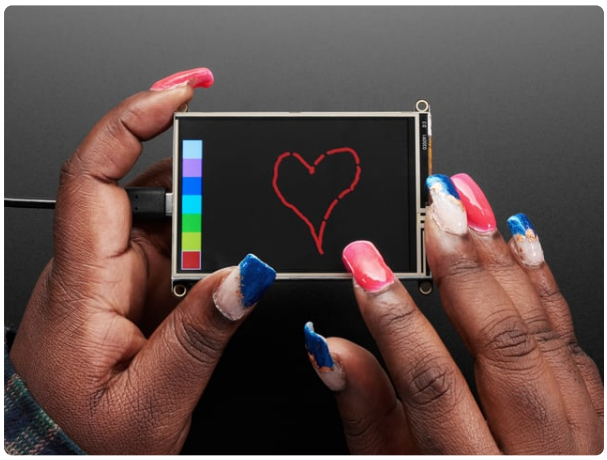
It offers an environment similar to the Linux terminal-based editor program Nano. It uses a CircuitPython implementation of a tiny subset of the Curses (https://adafru.it/19fk) library called "dang".

The code for this project was built on top of the work of jepler (https://adafru.it/19fl) which itself was forked from a Python version created by Wasim Lorgat (@seeM). The original CPython version of the editor came with a tutorial (https://adafru.it/19fm).

Hardware

The Feather RP2040 with USB Host is the main brain of the project. It's plugged into a 3.5" TFT Featherwing Display with 480x320 resolution, though other displays and sizes will work as well. I used the adorable little keyboard from the Adafruit shop, but any standard USB HID Keyboard can be used, so if you've already got a different favorite keyboard, just plug it in and type away!

Parts



Adafruit TFT FeatherWing - 3.5" 480x320 Touchscreen for Feathers

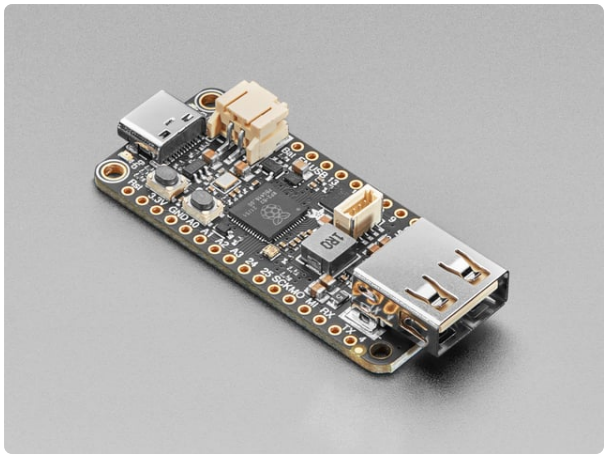
Spice up your Feather project with a beautiful 3.5" touchscreen display shield with built in microSD card socket. This TFT display is 3.5" diagonal with a bright 6 white-LED...

https://www.adafruit.com/product/3651

[Miniature Keyboard- Microcontroller-Friendly PS/2 and USB]

Add a typing interface to your project with this microcontroller-friendly miniature keyboard. We found the smallest PS/2+USB keyboard available, a mere 8.75" x 4.65" x...

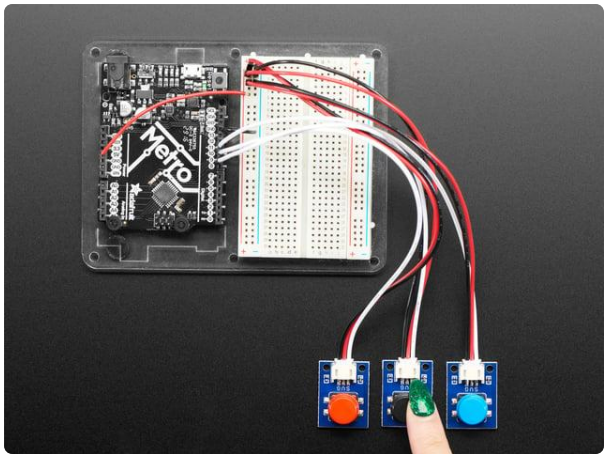
https://www.adafruit.com/product/857



[Adafruit Feather RP2040 with USB Type A Host]

You're probably really used to microcontroller boards with USB, but what about a dev board with two? Two is more than one, so that makes it twice as good! And...

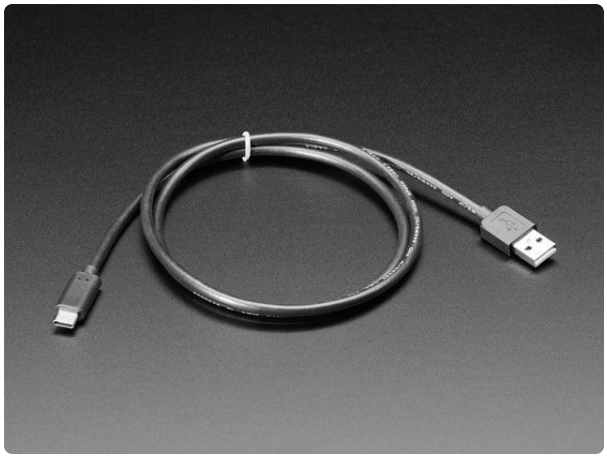
https://www.adafruit.com/product/5723



[STEMMA Wired Tactile Push-Button Pack - 5 Color Pack]

Little clicky switches are standard input "buttons" on electronic projects. These are just like our Colorful Round...

https://www.adafruit.com/product/4431



[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

Project Setup

Storage Writable Mode

If you want to be able to create new files, or save changes to existing ones, then you need the storage in your CircuitPython device to be mounted as writable. This can be achieved with a little bit of code inside of the **boot.py** file. For this project, I've hooked up a button that the user can press during boot up to configure the device as writable vs. read only. This technique and more general information about the storage module can be found on this CircuitPython Essentials guide page (https://adafru.it/DIE). If you're unfamiliar with that process, I recommend you look over that Essentials guide page first and then return here. The devices and specific pins mentioned on the Essentials page are different from the ones this project uses, but the technique is the exact same.

Loading **boot.py**

If you do not already have a **boot.py** file on your **CIRCUITPY** drive, you can simply copy the one from the project files onto your device. If you do have a pre-existing **boot.py** with some code that you want to keep you can copy/paste the code from the project **boot.py** file into your own existing file.

```
# SPDX-FileCopyrightText: 2024 Tim Cocks
#
# SPDX-License-Identifier: MIT
import usb_cdc
import board
import digitalio
import storage

usb_cdc.enable(console=True, data=True)    # Enable console and data

write_mode_btn = digitalio.DigitalInOut(board.D9)
write_mode_btn.direction = digitalio.Direction.INPUT
write_mode_btn.pull = digitalio.Pull.UP

storage.remount("/", readonly=write_mode_btn.value)
```

Load the Code

Click the Download Project Bundle button in the window below. It will download all the code for this project to your computer as a zipped folder.

After downloading the Project Bundle, plug your Feather into the computer's USB port with a known good USB data+power cable. You should see a new flash drive appear in the computer's File Explorer or Finder (depending on your operating system) called

CIRCUITPY. Unzip the folder and copy the following items to the Feather **CIRCUITPY** drive.

- **lib** folder
- **adafruit_editor** folder
- **code.py**

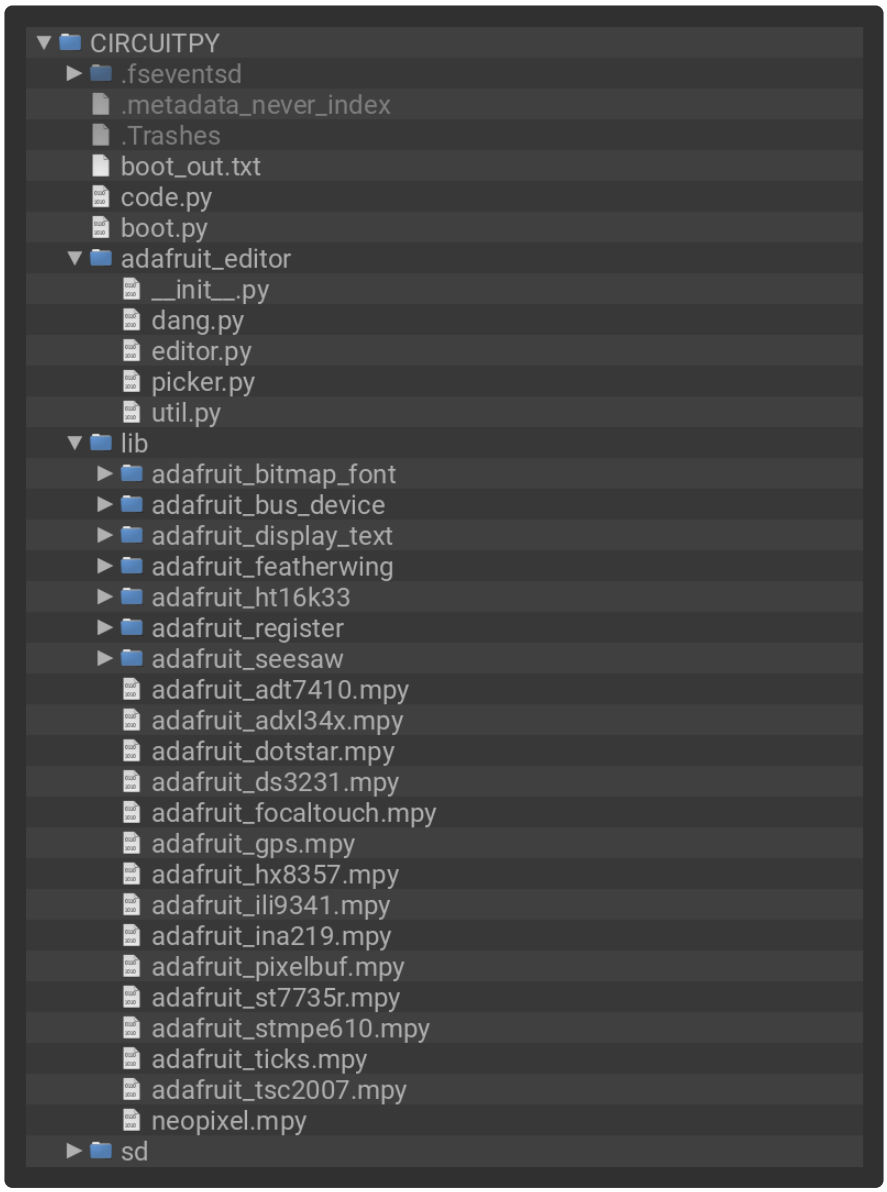
Code

The **code.py** file for the project is shown below:

```
# SPDX-FileCopyrightText: 2024 Tim Cocks
#
# SPDX-License-Identifier: MIT
import traceback
from adafruit_editor import editor, picker
from adafruit_featherwing import tft_featherwing_35
import terminalio
import displayio
from adafruit_display_text.bitmap_label import Label
import usb_cdc
#pylint: disable=redefined-builtin,broad-except

def print(message):
    usb_cdc.data.write(f"{message}\r\n".encode("utf-8"))


tft_featherwing = tft_featherwing_35.TFTFeatherWing35V2()
display = tft_featherwing.display
display.rotation = 180

customized_console_group = displayio.Group()
display.root_group = customized_console_group
customized_console_group.append(displayio.CIRCUITPYTHON_TERMINAL)

visible_cursor = Label(terminalio.FONT, text="",
                       color=0x000000, background_color=0xeeeeee, padding_left=1)
visible_cursor.hidden = True
visible_cursor.anchor_point = (0, 0)
customized_console_group.append(visible_cursor)

try:
    while True:
        try:
            visible_cursor.hidden = True
            filename = picker.pick_file()
        except KeyboardInterrupt:
            customized_console_group.remove(displayio.CIRCUITPYTHON_TERMINAL)
            break

        try:
            visible_cursor.hidden = False
            editor.edit(filename, visible_cursor)
        except KeyboardInterrupt:
            visible_cursor.hidden = True

# Any Exception, including Keyboard Interrupt
except Exception as e:
    print("\n".join(traceback.format_exception(e)))
    customized_console_group.remove(displayio.CIRCUITPYTHON_TERMINAL)
```

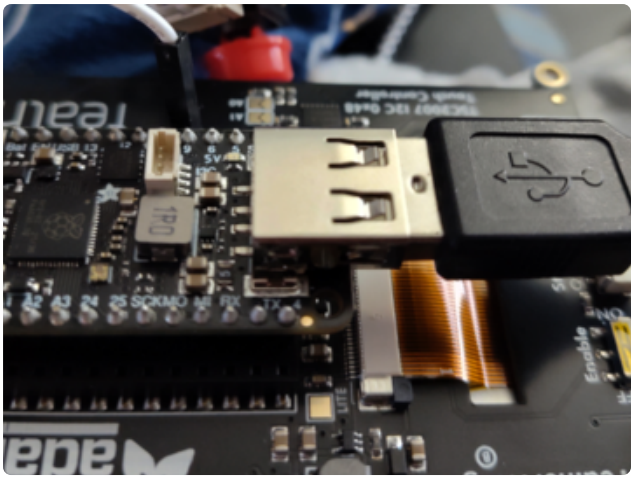
Drive Structure

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



USB Host Keyboard

Keyboard not working? If your keyboard is not working or the seems unresponsive when you are pressing buttons, unplug and then plug back in the keyboard to the USB Type A Host port on the Feather.



When the keyboard is plugged in, it will be automatically detected by CircuitPython and be able to be used to input code into the REPL or into calls to the `input()` function.

Editor Project Integration

The editor project is built to interact directly with the input (stdin) and output (stdout) streams within the core system. That means that once the keyboard is plugged in and recognized by the system, there is no other specific code needed to initialize, setup the keyboard, or poll for key presses. Conveniently, all of those just work™ with this project.

Code Walkthrough

Display Init

The display is initialized using the Adafruit Featherwing Library (https://adafru.it/19fn). If you're using a different display or the V1 revision of the TFT Featherwing, you'll need to adjust the initialization code accordingly.

```
from adafruit_featherwing import tft_featherwing_35
```

```
tft_featherwing = tft_featherwing_35.TFTFeatherWing35V2()
display = tft_featherwing.display

# Uncomment if you want to flip the display over
# display.rotation = 180
```

Showing CIRCUITPYTHON_TERMINAL

CircuitPython `displayio` core module includes `displayio.CIRCUITPYTHON_TERMINAL`, which is an internal `Group` object that contains the `terminalio.Terminal` that represents the visible console output which is shown on the display by default until you replace it with something else by setting the Display `root_group`.

Typically your code doesn't need to interact with it, but in this project we want to layer the visible cursor on top of it. Therefore we are going to take that built-in Group and add it to our own custom Group which we can then add the cursor `Label` things to in order to have them shown on top of the Terminal.

```
customized_console_group = displayio.Group()
display.root_group = customized_console_group
customized_console_group.append(displayio.CIRCUITPYTHON_TERMINAL)
```

One important thing to note is that you must remove the TERMINAL from your custom group at or before the time that your program exits, or else it can cause the system to crash when the core tries to show it on the Display. This project uses a very broad try/catch block to catch any possible errors including the Keyboard Interrupt raised when the user presses Ctrl-C. When any exception is raised, **code.py** removes the TERMINAL from its own custom Group, thereby freeing it back up to be shown by the core system.

```
try:
    # ... main program ...

# Any Exception, including Keyboard Interrupt
except Exception as e:
    print("\n".join(traceback.format_exception(e)))
    customized_console_group.remove(displayio.CIRCUITPYTHON_TERMINAL)
```

The exception is also printed using the `traceback` module so that you can see the stack trace pointing to whichever lines of code caused the exception.

Visible Cursor

The `terminalio.Terminal` class does internally keep track of a cursor position, but it doesn't currently have any way to make visible cursor on the display. This project uses a Label object from `adafruit_display_text` that is configured with opposite colors and contains only a single character at a time. This Label is appended into the

Group to be drawn layered on top of the Terminal and it gets moved around to the appropriate x, y coordinates whenever the the cursor position moves.

The visible cursor Label gets initialized in code.py and then passed as an argument to the `editor.edit()` function so that the editor can move it around internally as it's responding to input from the user.

```
# initialization
visible_cursor = Label(terminalio.FONT, text="", color=0x000000,
background_color=0xeeeeee, padding_left=1)
visible_cursor.hidden = True
visible_cursor.anchor_point = (0, 0)
customized_console_group.append(visible_cursor)

# ... main program loop begins ...

    editor.edit(filename, visible_cursor)
```

Helper Modules

Aside from **code.py** and **boot.py,** there are 4 helper modules used in the project source code. Here is a brief description of each class and how it gets used:

- **util.py** - Only contains a single function `readonly()` which other modules make use of. It returns True if the device is in readonly mode or False if it's writable.
- **dang.py** - A small subset of the Curses library from CPython. It has the `Screen` class which handles high level input and output to the standard in/out streams. The other modules use the `wrapper()` function to run within the Screen it creates.
- **picker.py** - The file chooser portion of the app. Shows a list of files on **CIRCUITPY**, allows the user to move up and down the list with arrow keys. The Enter button will select a file and launch the editor to open it. Ctrl-N will prompt the user for the name of a new file and then open up the editor to a new empty file of the name entered.
- **editor.py** - The file editing portion of the app. Shows the contents of a specified file and allows the user to interact with it by moving the cursor around and inserting or changing text within it. If the device is in writable mode, Ctrl-X will save and exit. Ctrl-C exits without saving.

The original CPython version of the editor came with a tutorial (https://adafru.it/19fm) which covers in more depth the editor code structure and modules used.

USB Data Device

If you simply want to use the project and nothing more, then you won't need to worry about this section. However if you are interested in modifying or developing further

the text editor from this project, the steps and information in this section will be very helpful during development and testing.

By default in CircuitPython when your code uses a statement like `print("something")`, the message is output into the serial console which is made visible by Mu, or can be connected to with Tio, PuTTY, or other serial programs. This is great for seeing the value of your variables or state of your program as you're developing and debugging it.

This editor project uses the standard output stream to show its GUI, so if we were to print things normally, they would get plastered onto our visible GUI, clobbering some portion of the file contents or list of files that happened to be unlucky enough to be in the same location that the print statement went to.

Luckily CircuitPython offers the option to enable a secondary USB serial data device in addition to standard console device. This learn guide page (https://adafru.it/SC9) discusses the data line and shows the code necessary to enable it. The code must be put into **boot.py** file.

```
usb_cdc.enable(console=True, data=True)    # Enable console and data
```

Once enabled, you can access the data serial device with `usb_cdc.data`. This object has a `write()` function that takes bytestrings and outputs them to the serial data device. It's often helpful to include newline escape characters at the end of your message so that it will print one message per line.

```
usb_cdc.data.write(f"Message Here\r\n".encode("utf-8"))
```

In this project's code files there is an overridden `print()` function which outputs to this data line. That means within the code whenever there is a `print()` statement, it will actually get printed out on the data line rather than the standard console line.

```
def print(message):
    usb_cdc.data.write(f"{message}\r\n".encode("utf-8"))
```

In order to see those messages, you'll need to connect to the USB serial data device separately from the standard serial console connection. I am using Linux and by default the two serial devices show up as:

- `/dev/ttyACM0` is the standard console serial line that I use to access normal code output and REPL.

- `/dev/ttyACM1` is the secondary data serial line that is used to access any messages sent via the data serial device. This port is used for development and testing of this project.

See the "Which Serial Port on the Host?" section in the Customizing USB Devices Learn Guide (https://adafru.it/SC9) to see how to find the serial ports on your computer.