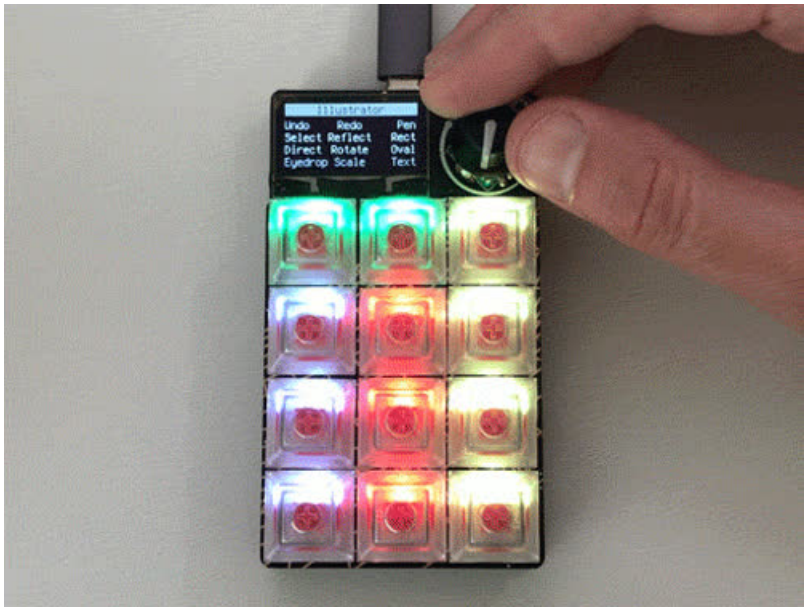
adafruit learning system

MACROPAD Hotkeys

Created by Phillip Burgess



Last updated on 2021-08-25 08:09:27 PM EDT

Guide Contents

Overview

When you have a key-festooned unit called a **MACROPAD**, it's only natural that one of the first things to try would be **application hotkeys** or **macros**. Anything less would be like a *dinosaur tour without any dinosaurs!*

Press one of MACROPAD's 12 keys to send a shortcut, function key or whole sequence of keystrokes to a connected computer. The OLED display provides a map, while LEDs under each key offer color-coded groups or themes. Turn the dial to select among different application sets.

This is one of those projects that you can simply find everyday use for as-is, or peer inside the code to see how CircuitPython makes this all pretty simple. Additionally, hotkey **configuration files** for different desktop applications are easily **created, modified and shared**.

Parts

You can buy a kit with all the parts with Kailh Red keys and clear keycaps or build your own custom configuration:

Your browser does not support the video tag.

Adafruit MacroPad RP2040 Starter Kit - 3x4 Keys + Encoder + OLED

Strap yourself in, we're launching in T-minus 10 seconds...Destination? A new Class M planet called MACROPAD! M here stands for Microcontroller because this 3x4 keyboard...

$49.95

In Stock

Add to Cart

- or -

Adafruit MACROPAD RP2040 Bare Bones - 3x4 Keys + Encoder + OLED

Strap yourself in, we're launching in T-minus 10 seconds...Destination? A new Class M planet called MACROPAD! M here, stands for Microcontroller because this 3x4 keyboard...

$34.95

In Stock

Add to Cart

Adafruit MacroPad RP2040 Enclosure + Hardware Add-on Pack

Dress up your Adafruit Macropad with PaintYourDragon's fabulous decorative silkscreen enclosure and hardware kit. You get the two custom PCBs that are cut to act as a protective...

$4.95

In Stock

Add to Cart

Kailh Mechanical Key Switches - 10 packs - Cherry MX Compatible

For crafting your very own custom keyboard, these Kailh mechanical key switches are deeee-luxe!Come in a pack of 10 switches, plenty to make a...

Out of Stock

Add from Store

Clear DSA Keycaps for MX Compatible Switches - 10 pack

Get ready to clacky to your heart's content. Here is a 10 pack of clear transparent DSA keycaps for your next mechanical keyboard or

$5.95

In Stock

Add to Cart

Translucent Keycaps for MX Compatible Switches - 10 pack

Get ready to clacky to your heart's content. Here is a 10 pack of translucent keycaps for your next mechanical keyboard or

$4.95

In Stock

Add to Cart

CircuitPython

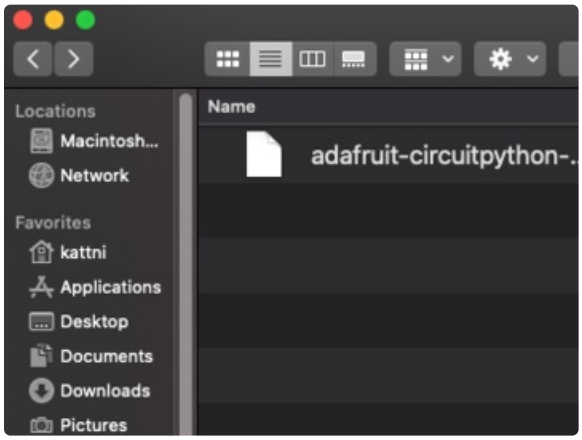
CircuitPython (https://adafru.it/tB7) is a derivative of MicroPython (https://adafru.it/BeZ) designed to simplify experimentation and education on low-cost microcontrollers. It makes it easier than ever to get prototyping by requiring no upfront desktop software downloads. Simply copy and edit files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

CircuitPython Quickstart

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

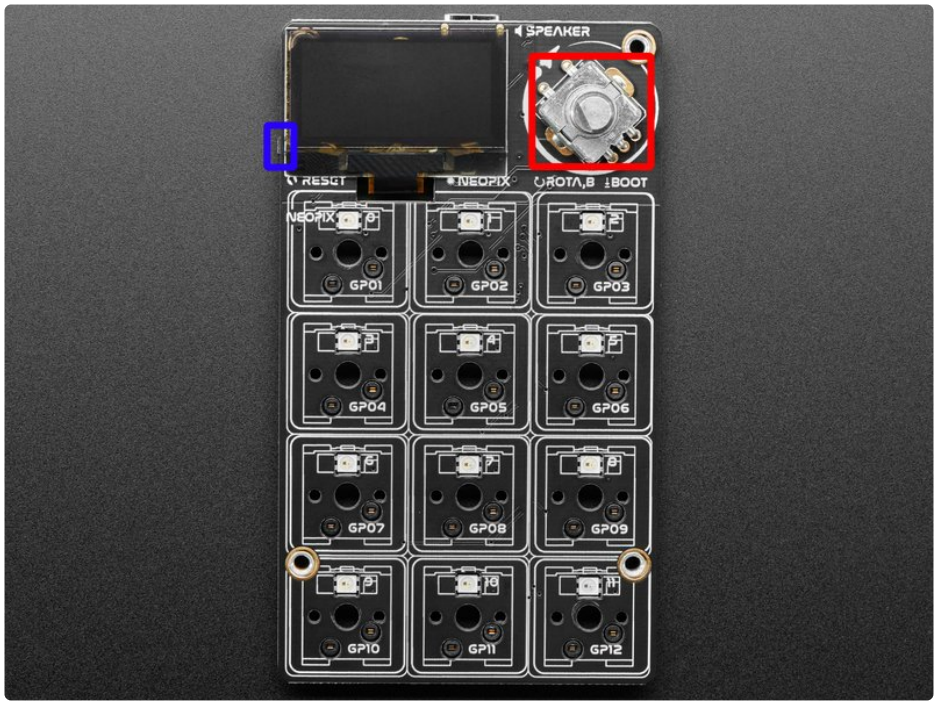
https://adafru.it/TB9

https://adafru.it/TB9



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

Save it wherever is convenient for you.



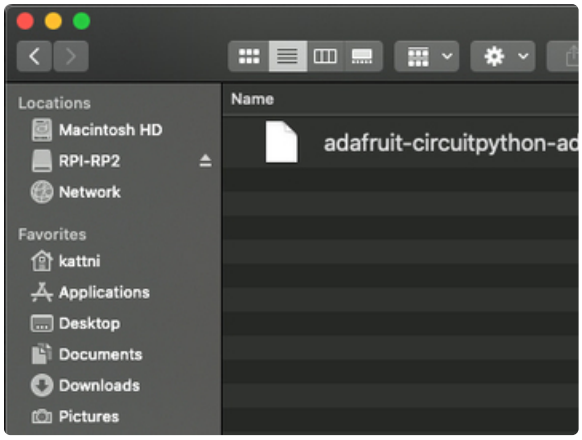
The BOOT button is the button switch in the rotary encoder! To engage the BOOT button, simply press down on the rotary encoder.

To enter the bootloader, hold down the **BOOT/BOOTSEL button** (highlighted in red above), and while continuing to hold it (don't let go!), press and release the **reset button** (highlighted in blue above). **Continue to hold the BOOT/BOOTSEL button until the RPI-RP2 drive appears!**

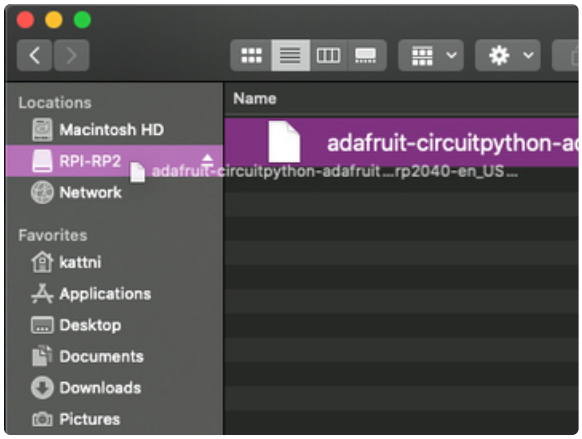
If the drive does not appear, release all the buttons, and then repeat the process above.

You can also start with your board unplugged from USB, press and hold the BOOTSEL button (highlighted in red above), continue to hold it while plugging it into USB, and wait for the drive to appear before releasing the button.

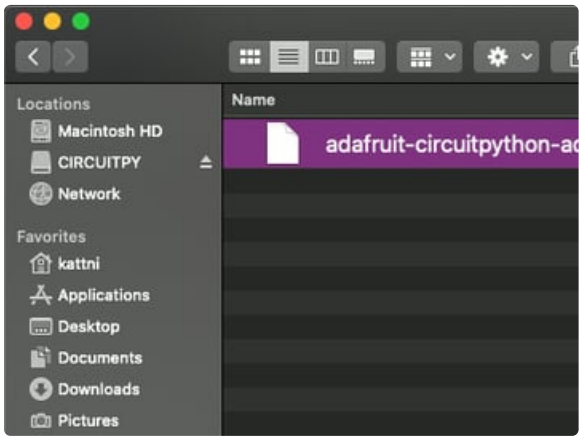
A lot of people end up using charge-only USB cables and it is very frustrating! **Make sure you have a USB cable you know is good for data sync.**



You will see a new disk drive appear called **RPI-RP2**.



Drag the **adafruit_circuitpython_etc.uf2** file to **RPI-RP2.**



The **RPI-RP2** drive will disappear and a new disk drive called **CIRCUITPY** will appear.

That's it, you're done! :)

Safe Mode

You want to edit your **code.py** or modify the files on your **CIRCUITPY** drive, but find that you can't. Perhaps your board has gotten into a state where **CIRCUITPY** is read-only. You may have turned off the **CIRCUITPY** drive altogether. Whatever the reason, safe mode can help.

Safe mode in CircuitPython does not run any user code on startup, and disables auto-reload. This means a few things. First, safe mode *bypasses any code in* **boot.py** (where you can set **CIRCUITPY** read-only or turn it off completely). Second, *it does not run the code in* **code.py**. And finally, *it does not automatically soft-reload when data is written to the* **CIRCUITPY** *drive*.

Therefore, whatever you may have done to put your board in a non-interactive state, safe mode gives you the opportunity to correct it without losing all of the data on the **CIRCUITPY** drive.

Entering Safe Mode in CircuitPython 6.x

This section explains entering safe mode on CircuitPython 6.x.

To enter safe mode when using CircuitPython 6.x, plug in your board or hit reset (highlighted in red above). Immediately after the board starts up or resets, it waits 700ms. On some boards, the onboard status LED (highlighted in green above) will turn solid yellow during this time. If you press reset during that 700ms, the board will start up in safe mode. It can be difficult to react to the yellow LED, so you may want to think of it simply as a slow double click of the reset button. (Remember, a fast double click of reset enters the bootloader.)

Entering Safe Mode in CircuitPython 7.x

This section explains entering safe mode on CircuitPython 7.x.

To enter safe mode when using CircuitPython 7.x, plug in your board or hit reset (highlighted in red above). Immediately after the board starts up or resets, it waits 1000ms. On some boards, the onboard status LED (highlighted in green above) will blink yellow during that time. If you press reset during that 1000ms, the board will start up in safe mode. It can be difficult to react to the yellow LED, so you may want to think of it simply as a slow double click of the reset button. (Remember, a fast double click of reset enters the bootloader.)

In Safe Mode

Once you've entered safe mode successfully in CircuitPython 6.x, the LED will pulse yellow.

If you successfully enter safe mode on CircuitPython 7.x, the LED will intermittently blink yellow three times.

If you connect to the serial console, you'll find the following message.

```
Auto-reload is off.
Running in safe mode! Not running saved code.

CircuitPython is in safe mode because you pressed the reset button during boot. Press again to
exit safe mode.

Press any key to enter the REPL. Use CTRL-D to reload.
```

You can now edit the contents of the **CIRCUITPY** drive. Remember, *your code will not run until you press the reset button, or unplug and plug in your board, to get out of safe mode.*

Flash Resetting UF2

If your board ever gets into a really *weird* state and doesn't even show up as a disk drive when installing CircuitPython, try loading this 'nuke' UF2 which will do a 'deep clean' on your Flash Memory. **You will lose all the files on the board**, but at least you'll be able to revive it! After loading this UF2, follow the steps above to re-install CircuitPython.

https://adafru.it/RLE

https://adafru.it/RLE

Project Code

Text Editor

Adafruit recommends using the **Mu** editor for editing your CircuitPython code. You can get more info in this guide (https://adafru.it/ANO).

Alternatively, you can use any text editor that saves simple text files.

Download the Project Bundle

Your project will use a specific set of CircuitPython libraries and the **code.py** file, along with a folder full of key configuration files. To get everything you need, click on the **Download Project Bundle** link below, and uncompress the .zip file.

Drag the contents of the uncompressed bundle directory onto your MACROPAD board's **CIRCUITPY** drive, replacing any existing files or directories with the same names, and adding any new ones that are necessary.

Inside the **macros** folder, you'll likely want to pluck out any macro settings files to discard. They're prefixed with "mac" and "win"for MacOS and Windows since these systems have different modifier keys. These files are just a starting point for reference...more likely, you'll start creating your own.

If updating from a prior version of this code, move any key configuration files that you've created or edited to a safe place first, so they're not overwritten or lost, then move them back to the CIRCUITPY/macros folder after updating.

```
"""
A fairly straightforward macro/hotkey program for Adafruit MACROPAD.
Macro key setups are stored in the /macros folder (configurable below),
load up just the ones you're likely to use. Plug into computer's USB port,
use dial to select an application macro set, press MACROPAD keys to send
key sequences.
"""

# pylint: disable=import-error, unused-import, too-few-public-methods

import os
import time
import displayio
import terminalio
from adafruit_display_shapes.rect import Rect
from adafruit_display_text import label
from adafruit_macropad import MacroPad


# CONFIGURABLES ------------------------
```

```
MACRO_FOLDER = '/macros'


# CLASSES AND FUNCTIONS ----------------

class App:
    """ Class representing a host-side application, for which we have a set
        of macro sequences. Project code was originally more complex and
        this was helpful, but maybe it's excessive now?"""
    def __init__(self, appdata):
        self.name = appdata['name']
        self.macros = appdata['macros']

    def switch(self):
        """ Activate application settings; update OLED labels and LED
            colors. """
        group[13].text = self.name   # Application name
        for i in range(12):
            if i < len(self.macros): # Key in use, set label + LED color
                macropad.pixels[i] = self.macros[i][0]
                group[i].text = self.macros[i][1]
            else:  # Key not in use, no label or LED
                macropad.pixels[i] = 0
                group[i].text = ''
        macropad.keyboard.release_all()
        macropad.pixels.show()
        macropad.display.refresh()


# INITIALIZATION -----------------------

macropad = MacroPad()
macropad.display.auto_refresh = False
macropad.pixels.auto_write = False

# Set up displayio group with all the labels
group = displayio.Group()
for key_index in range(12):
    x = key_index % 3
    y = key_index // 3
    group.append(label.Label(terminalio.FONT, text='', color=0xFFFFFF,
                             anchored_position=((macropad.display.width - 1) * x / 2,
                                                macropad.display.height - 1 -
                                                (3 - y) * 12),
                             anchor_point=(x / 2, 1.0)))
group.append(Rect(0, 0, macropad.display.width, 12, fill=0xFFFFFF))
group.append(label.Label(terminalio.FONT, text='', color=0x000000,
                         anchored_position=(macropad.display.width//2, -2),
                         anchor_point=(0.5, 0.0)))
macropad.display.show(group)

# Load all the macro key setups from .py files in MACRO_FOLDER
apps = []
files = os.listdir(MACRO_FOLDER)
files.sort()
for filename in files:
    if filename.endswith('.py'):
```

```
    if filename.endswith('.py'):
        try:
            module = __import__(MACRO_FOLDER + '/' + filename[:-3])
            apps.append(App(module.app))
        except (SyntaxError, ImportError, AttributeError, KeyError, NameError,
                IndexError, TypeError) as err:
            pass

if not apps:
    group[13].text = 'NO MACRO FILES FOUND'
    macropad.display.refresh()
    while True:
        pass

last_position = None
last_encoder_switch = macropad.encoder_switch_debounced.pressed
app_index = 0
apps[app_index].switch()


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

while True:
    # Read encoder position. If it's changed, switch apps.
    position = macropad.encoder
    if position != last_position:
        app_index = position % len(apps)
        apps[app_index].switch()
        last_position = position

    # Handle encoder button. If state has changed, and if there's a
    # corresponding macro, set up variables to act on this just like
    # the keypad keys, as if it were a 13th key/macro.
    macropad.encoder_switch_debounced.update()
    encoder_switch = macropad.encoder_switch_debounced.pressed
    if encoder_switch != last_encoder_switch:
        last_encoder_switch = encoder_switch
        if len(apps[app_index].macros) < 13:
            continue    # No 13th macro, just resume main loop
        key_number = 12 # else process below as 13th macro
        pressed = encoder_switch
    else:
        event = macropad.keys.events.get()
        if not event or event.key_number >= len(apps[app_index].macros):
            continue # No key events, or no corresponding macro, resume loop
        key_number = event.key_number
        pressed = event.pressed

    # If code reaches here, a key or the encoder button WAS pressed/released
    # and there IS a corresponding macro available for it...other situations
    # are avoided by 'continue' statements above which resume the loop.

    sequence = apps[app_index].macros[key_number][2]
    if pressed:
        # the sequence is arbitrary-length
        # each item in the sequence is either
        # an integer (e.g., Keycode.KEYPAD_MINUS),
```

```
        # a floating point value (e.g., 0.20)
        # or a string.
        # Positive Integers ==> key pressed
        # Negative Integers ==> key released
        # Float             ==> sleep in seconds
        # String            ==> each key in string pressed & released
        if key_number < 12: # No pixel for encoder button
            macropad.pixels[key_number] = 0xFFFFFF
            macropad.pixels.show()
        for item in sequence:
            if isinstance(item, int):
                if item >= 0:
                    macropad.keyboard.press(item)
                else:
                    macropad.keyboard.release(-item)
            elif isinstance(item, float):
                time.sleep(item)
            else:
                macropad.keyboard_layout.write(item)
    else:
        # Release any still-pressed keys
        for item in sequence:
            if isinstance(item, int) and item >= 0:
                macropad.keyboard.release(item)
        if key_number < 12: # No pixel for encoder button
            macropad.pixels[key_number] = apps[app_index].macros[key_number][0]
            macropad.pixels.show()
```

Custom Configurations

You can add or remove MACROPAD configurations for different applications just by moving files in or out of the **CIRCUITPY/macros** folder. At its simplest, you can collect existing configuration files and never need to edit anything.

Each of these files is really just a snippet of **CircuitPython** code. They can be modified with any text editor, and text files are easily shared, for example in the Adafruit Forums (https://adafru.it/jlf).

You could start by copying one of the examples in the **macros** folder. Give it a descriptive name...and, if you'll be sharing this with others, consider mentioning right in the filename what system it's for, since key sequences vary among platforms (e.g. COMMAND vs. CONTROL on Mac vs Windows).

Here's one of the examples, **mac-safari.py**, for the Safari web browser on MacOS:

```
Temporarily unable to load content:
```

This is just a single Python dictionary (which **must** be named '**app**' for the project code to find it) containing two keys: '**name**' and '**macros**'.

'**name**' is what's shown across the top of MACROPAD's display when turning the knob to switch settings. This must be a short string in quotes, no more than 20 characters, for example **'Safari'**.

'**macros**' is a list [enclosed in square brackets] of tuples (each enclosed in parenthesis) — one for each of MACROPAD's 12 keys, and optionally one more for the encoder button. These appear sequentially, first item for the top-left key, second for top-center, and so forth. The examples have some Python comments to explain what's happening.

Each of these tuples contains **three** elements:

1. A **hexadecimal RGB color** value for the corresponding **LED**. You may want to avoid bright values, as a whole keypad of these can be distracting. If you like, color-code related groups of keys, or theme whole applications so you can tell what's active at a glance.
2. A brief **description**, in quotes. This is what's displayed on MACROPAD's OLED screen. *BRIEF* is the operative word here, ideally **six characters or less**. You can *sometimes* sneak in a 7-character label if adjoining items are shorter.
3. A **key sequence**, which can either be:
    1. A **single character or a string, in quotes** , if the key sequence is just regular keypresses (including SHIFT for uppercase characters).
    2. A **list [enclosed in square brackets]** of key code constants and quoted-strings, which will be issued in-order.

Most letters, numbers and symbols are best done as quoted strings...but for special navigation-type keys... arrows, COMMAND/CONTROL/OPTION, function keys and so forth...key constants are required. We can

see a complete list of these by typing in the CircuitPython REPL:

```
>>> from adafruit_hid.keycode import Keycode
>>> print(dir(Keycode))
['__class__', '__module__', '__name__', '__qualname__', '__bases__', '__dict__', 'C', 'M', 'A',
'B', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V',
'W', 'X', 'Y', 'Z', 'ONE', 'TWO', 'THREE', 'FOUR', 'FIVE', 'SIX', 'SEVEN', 'EIGHT', 'NINE',
'ZERO', 'ENTER', 'RETURN', 'ESCAPE', 'BACKSPACE', 'TAB', 'SPACEBAR', 'SPACE', 'MINUS', 'EQUALS',
'LEFT_BRACKET', 'RIGHT_BRACKET', 'BACKSLASH', 'POUND', 'SEMICOLON', 'QUOTE', 'GRAVE_ACCENT',
'COMMA', 'PERIOD', 'FORWARD_SLASH', 'CAPS_LOCK', 'F1', 'F2', 'F3', 'F4', 'F5', 'F6', 'F7',
'F8', 'F9', 'F10', 'F11', 'F12', 'PRINT_SCREEN', 'SCROLL_LOCK', 'PAUSE', 'INSERT', 'HOME',
'PAGE_UP', 'DELETE', 'END', 'PAGE_DOWN', 'RIGHT_ARROW', 'LEFT_ARROW', 'DOWN_ARROW', 'UP_ARROW',
'KEYPAD_NUMLOCK', 'KEYPAD_FORWARD_SLASH', 'KEYPAD_ASTERISK', 'KEYPAD_MINUS', 'KEYPAD_PLUS',
'KEYPAD_ENTER', 'KEYPAD_ONE', 'KEYPAD_TWO', 'KEYPAD_THREE', 'KEYPAD_FOUR', 'KEYPAD_FIVE',
'KEYPAD_SIX', 'KEYPAD_SEVEN', 'KEYPAD_EIGHT', 'KEYPAD_NINE', 'KEYPAD_ZERO', 'KEYPAD_PERIOD',
'KEYPAD_BACKSLASH', 'APPLICATION', 'POWER', 'KEYPAD_EQUALS', 'F13', 'F14', 'F15', 'F16', 'F17',
'F18', 'F19', 'LEFT_CONTROL', 'CONTROL', 'LEFT_SHIFT', 'SHIFT', 'LEFT_ALT', 'ALT', 'OPTION',
'LEFT_GUI', 'GUI', 'WINDOWS', 'COMMAND', 'RIGHT_CONTROL', 'RIGHT_SHIFT', 'RIGHT_ALT',
'RIGHT_GUI', 'modifier_bit']
```

The Safari example was chosen because it demonstrates most permutations quite nicely...

The third key, “Up,” for example...in Safari, this is done with SHIFT+SPACE. There’s no such thing as an “uppercase space,” so we can’t just use a quoted string here. **Keycode.SHIFT** is a constant telling the code to press and hold the SHIFT key, and then the quoted space is issued. SHIFT is automatically released at the end of the sequence.

You can see other keys doing similar operations, sometimes with **Keycode.COMMAND** or sometimes multiple modifiers (“Previous Tab,” for example, is CONTROL+SHIFT+TAB).

The bottom three keys show how to press *and release* keys mid-sequence. Each of these opens a new window (COMMAND+n) and enters a URL. We want the COMMAND key released after pressing 'n', so the characters in the URL string type normally, not as a list of commands. A *negative* Keycode value is used to indicate “release this key now”: **-Keycode.COMMAND** in the example.

Instead of a Keycode, a **floating-point number** inserts a **pause** in the key-pressing sequence. The duration is in seconds, either whole or partial. For example:

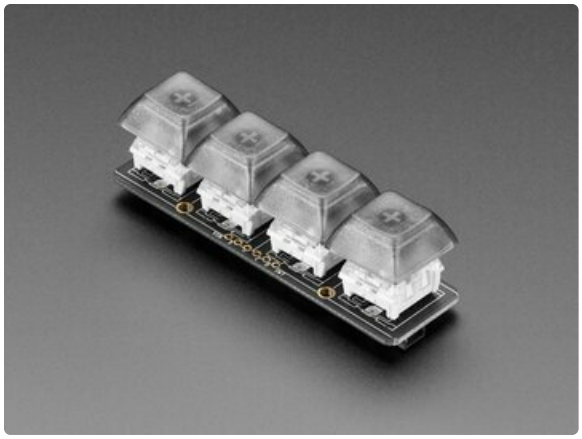
```
[Keycode.A, 0.5, Keycode.B, 1.0, Keycode.C]
```

This would type a lowercase “a,” pause one half second, type a “b,” pause one second, then type a “c.” Even if pausing for whole seconds, the decimal *must* be included...this is how it distinguishes from Keycodes, which are integer values.

Though you could *store passwords in there, this is* ***strongly discouraged****, since the CircuitPython code is not protected or secure. Anyone with access to your MACROPAD can read these files!*

Going Further

Once it's all started up, the code has little work to do...just watch for key press events and encoder movement, taking some action when necessary.



If one were so inclined, the hardware and code could be extended for extra functionality. MACROPAD features a Stemma QT connector on the side, which might accommodate some interesting sensors or a NeoKey 1x4 QT (https://adafru.it/TDb) for four extra key switches. The code as written does *not* handle any of this automatically...the additions would be a software project of your own.

Our hotkeys code uses MACROPAD's encoder wheel to select among different application settings. It's usually fine, but if you're a big multitasker it can be awkward when the wrong app is selected (which is why color-coding with the LEDs is recommended, for at-a-glance familiarity).

Could there be some way to *automatically* switch based on the current application in use? CircuitPython can receive serial messages while also emulating a keyboard, so there's ways to send information to MACROPAD. The host-side implementation though, *that* gets complex, and would vary with all the myriad system types and their particular scripting or development options, which is why it's not done here. Food for thought!