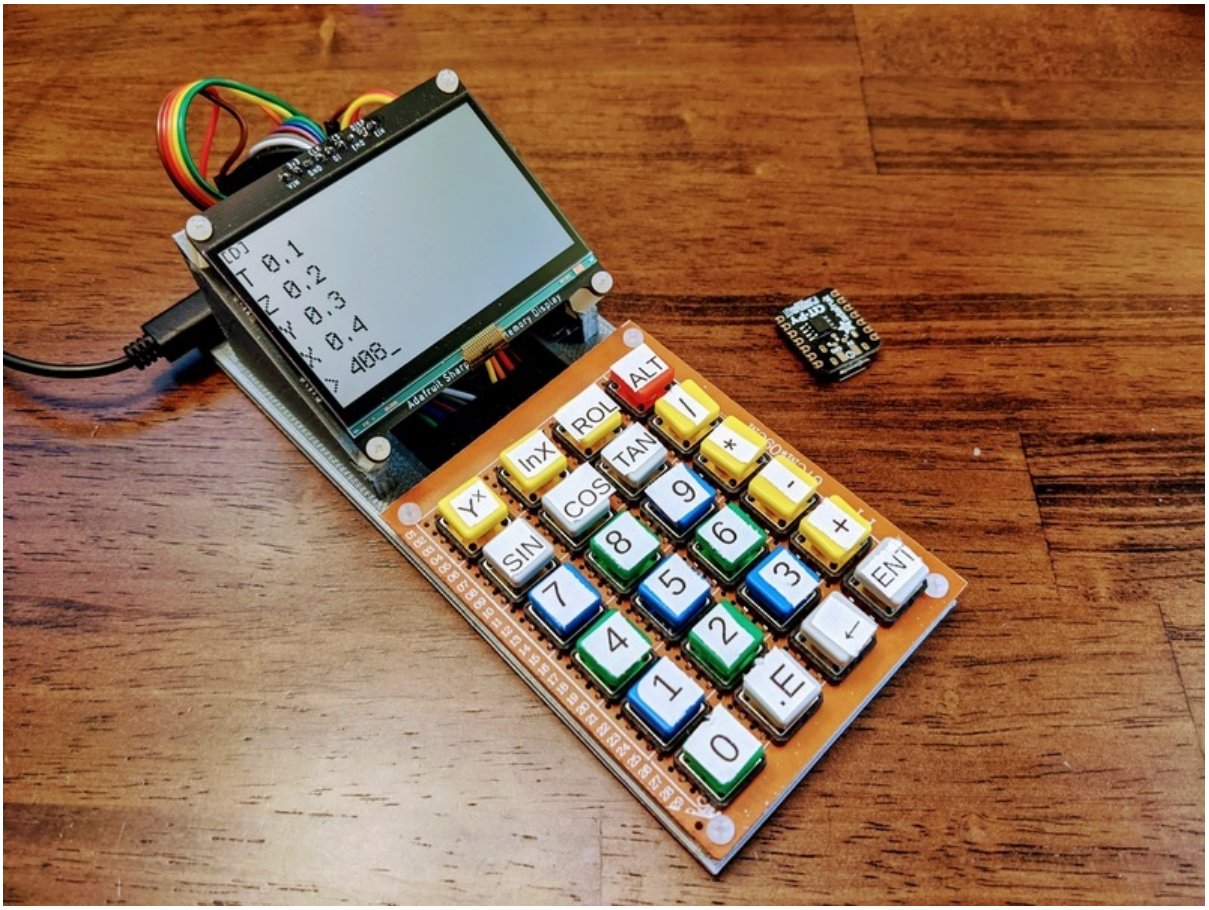
DIY Desktop Calculator with CircuitPython

Created by Jeff Epler



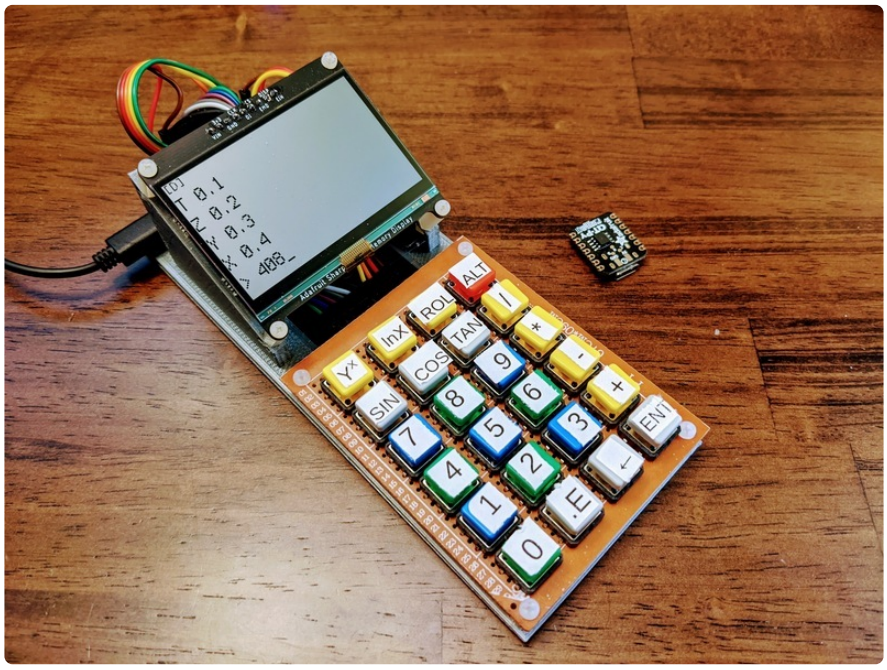
https://learn.adafruit.com/diy-rpn-desktop-calculator-with-circuitpython

Last updated on 2024-06-03 03:11:31 PM EDT

Table of Contents

Overview

In this project, you'll build your own Desktop Calculator with CircuitPython. Along the way you'll also learn about the alternative "RPN" notation for calculating, how a keyboard matrix works (and how to create your own!), and more!



What is "RPN", anyway?

Most of us are accustomed to standard mathematical notation, like 2+3*7 or (2+3)*7. Because most operations (like "+") appear between the two things operated on (like 2 and 3), this is called "infix" notation. Two alternatives to "infix" are "postfix" and "prefix". Postfix notation is also called RPN, which stands for "Reverse Polish Notation".

Possibly because it was used in some classic HP calculators, many people have a fondness for RPN or even feel that it works better than standard notation.

In standard infix notation, you have to know the "precedence rule", and understand that when given 2+3*7 you first compute 3*7=21 and then compute 2+21=23. When the rule does not reflect your intent, you add parentheses: (2+3)*7. In this case you evaluate 2+3=5 first, then 5*7=35. Mnemonic devices such as PEDMAS help us remember the rules.

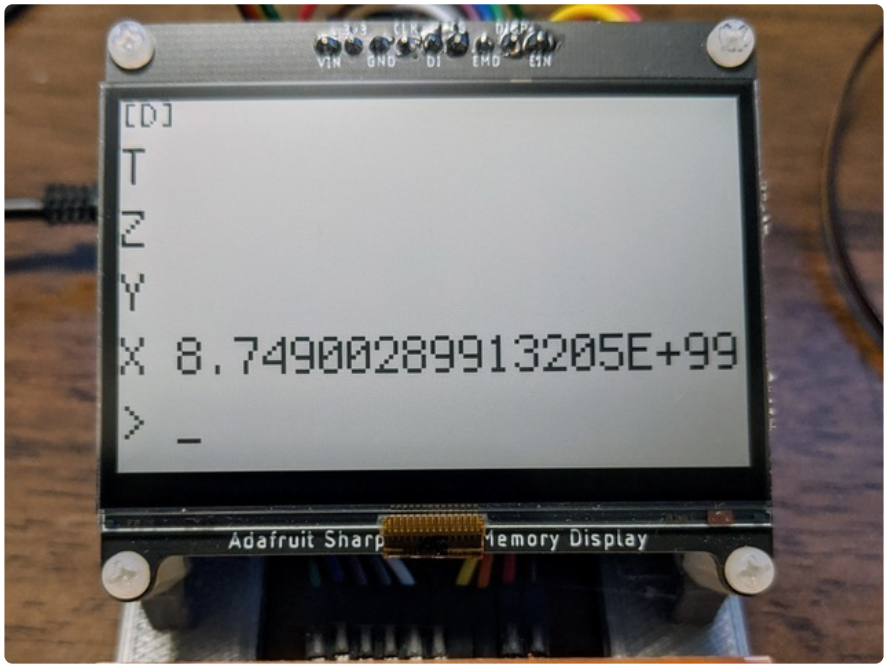
In postfix notation, there are no parentheses and no rules of precedence. Instead, there what is called the "stack". When you enter a number, it is placed on the stack. When you enter an operator such as "+" that operates on two values, the "top" two

values are taken off the stack, the operation is performed, and the result is put back on the stack.

If you'd like to know more about RPN, I found that the short paper "Order of Operations and RPN (https://adafru.it/QEx)" provides helpful background.

To interpret "2 3 7 * +", start with the empty stack and place each number on the stack as you encounter it. When you encounter "*", your stack will contain "2", "3", and "7" in that order. This makes the values that "+" will operate on "3" and "7", so they are removed and replaced with "21". Now the stack contains "2" and "21" in that order. When you encounter "+", you take "2" and "21" off and put "23" on. That's the same as "2 + 3 * 7" in standard notation.

Operations don't have to appear at the end only, they can appear in the middle. Consider "2 3 + 7 *". When you encounter "+", the stack has "2" and "3", so replace it with "5". Next, 7 gets placed on the stack. When you encouter "*", the stack has "5" and "7" so replace them with "35". That's the same as "(2 + 3) * 7" in standard notation.



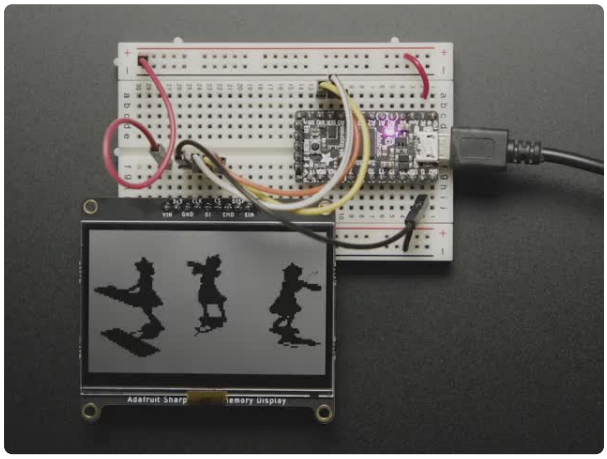
What is Decimal Arithmetic?

This calculator uses a decimal arithmetic library ported from standard Python3. It's available in the community bundle under the name jepler_udecimal. Decimal is "designed with people in mind, and necessarily has a paramount guiding principle– computers must provide an arithmetic that works in the same way as the arithmetic that people learn at school."

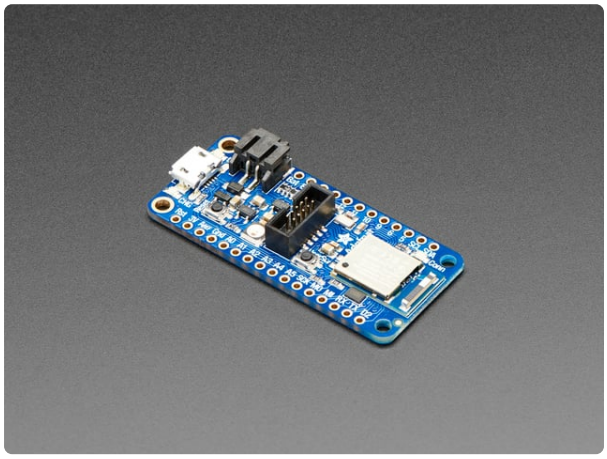
In this library, "0.1 + 0.2" is exactly equal to "0.3". Even better, the precision isn't limited to just 6 digits. In Python code, you can select any number of digits (even in the hundreds!). For this project, so that numbers fit on the display, numbers up to 14 digits are displayed.

This stands in contrast with how floats work in CircuitPython. CircuitPython flating point numbers only have about 6 to 7 digits of decimal precision, and values like "0.1" are not exact values.

Parts



Adafruit SHARP Memory Display Breakout - 2.7" 400x240 Monochrome
The Adafruit 2.7" 400x240 SHARP Memory Display Breakout is a chonky cross between an eInk (e-paper) display and an LCD. It has the...
https://www.adafruit.com/product/4694

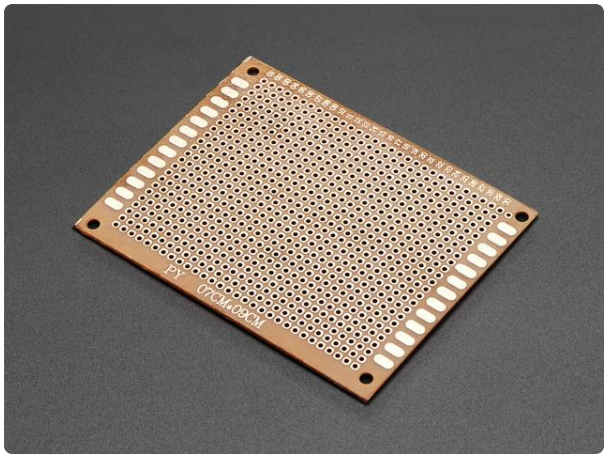


Adafruit Feather nRF52840 Express
The Adafruit Feather nRF52840 Express is the new Feather family member with Bluetooth Low Energy and native USB support featuring the nRF52840! It's...
https://www.adafruit.com/product/4062

Colorful 12mm Square Tactile Button Switch Assortment - 15 pack

Little clicky switches are standard input "buttons" on electronic projects. These work best in a PCB but can be...

https://www.adafruit.com/product/1010



Bakelite Universal Perfboard Plates - Pack of 10

Make your next project as you imagine it with prototyping perfboards that can easily be cut with scissors like these Bakelite Universal Perfboard Plates!We...

https://www.adafruit.com/product/2670

1 x Break-away 0.1" 36-pin strip right-angle male header (10 pack) https://www.adafruit.com/product/1540

Pack of ten

---

1 x Premium Female/Female Jumper Wires - 20 x 3" (75mm) Premium Female/Female Jumper Wires - 20 x 3" (75mm) https://www.adafruit.com/product/1951

Premium jumper wires (10 pack)

---

1 x White Nylon Screw and Stand-off Set – M2.5 Thread https://www.adafruit.com/product/3658

420 pieces

---

1 x Hook-up Wire Spool Set - 22AWG Solid Core - 6 x 25 ft https://www.adafruit.com/product/1311

Perfect for bread-boarding, 6 spools of solid-core wire

---

1 x Solder Wire - 60/40 Rosin Core - 0.5mm/0.02" diameter - 50 grams https://www.adafruit.com/product/1886

0.1 lb (about 50 grams) spool

Tools

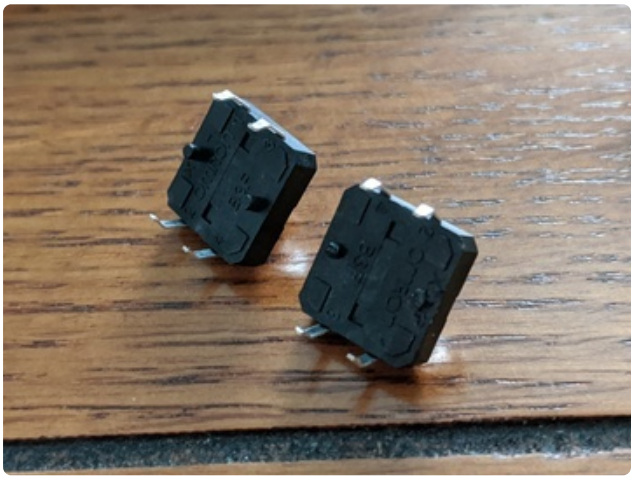
1 x Adjustable 30W 110V soldering iron https://www.adafruit.com/product/180
'pen-style' soldering iron

---

1 x Flush diagonal cutters https://www.adafruit.com/product/152
CHP170

Soldering the Keyboard Matrix

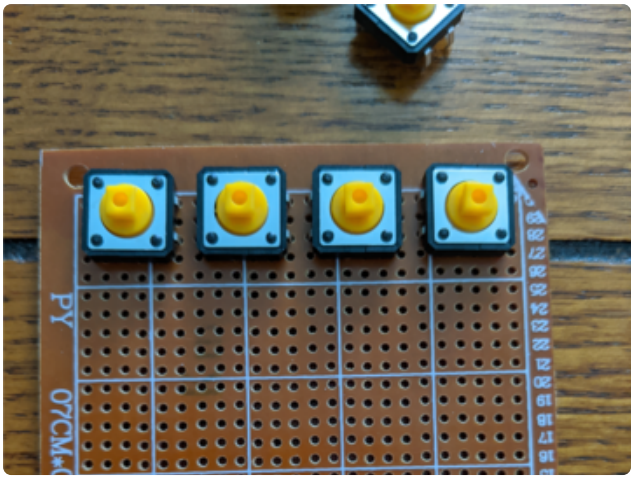
You'll need a total of 24 "12mm" through-hole tactile switches, one piece of 7x9CM perfboard, and preferably two colors of wire. If you buy 4 sets of Adafruit's colorful assortment, you'll have enough caps of a single color to do all the digits, with some left over for your next project.

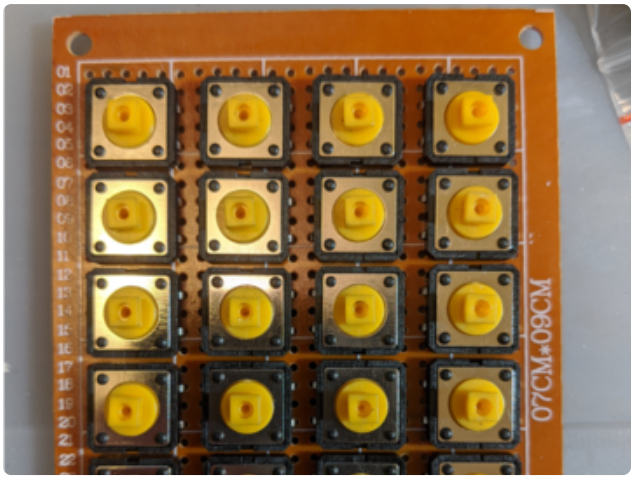


Begin by snipping the small plastic pins off of the bottom of each key. These are great if you're doing a custom PCB, don't hurt if you are using a protoboard, but just don't fit in the perfboard. You don't have to get them flush, just do the best you can with a pair of flush cutters.

The pins will fly off if you're not careful, so doing this directly in a small wastebasket is highly recommended!

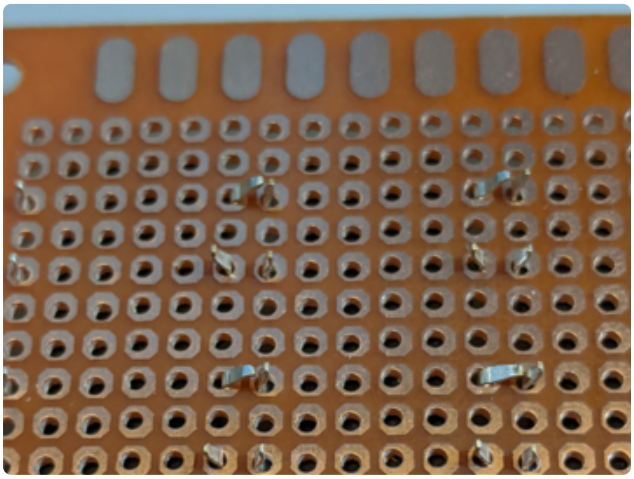
Use eye protection when using diagonal cutters and soldering to avoid eye injury.





Identify which side of the perfboard is the bottom—this is the side with the copper pads. We're going to stick the keys through from the top, non-copper side.

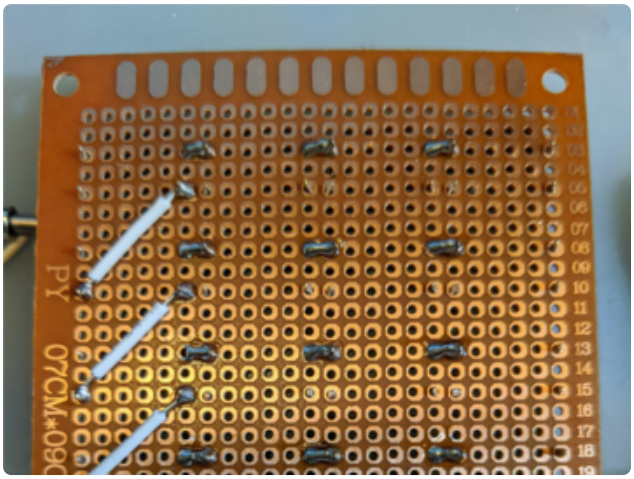
Insert the keys in the perfboard in a "4x6" (4 columns, 6 rows) pattern, with the pins sticking out the left and right sides. Use the very first row of holes (this will become the bottom of the keypad), and place everything as close as possible. This should leave you with an unused row at the top where we will later add the connecting header.

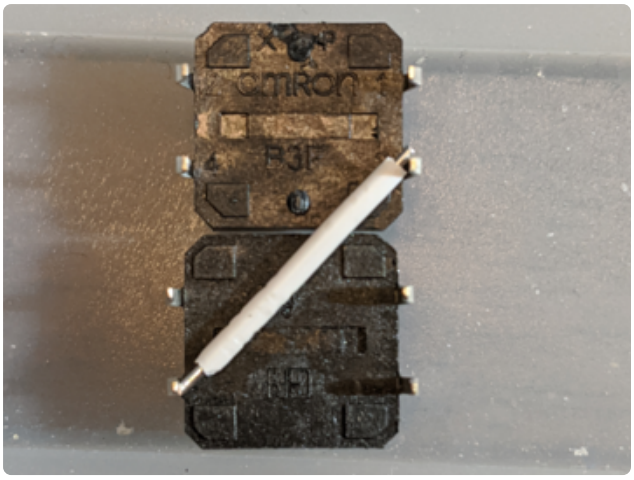


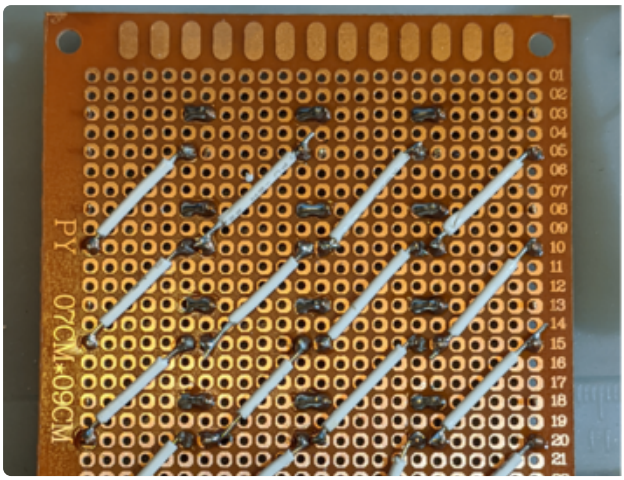
Flip the board over, the next task is to complete the "row wiring". There is already an internal connection between pins 1 and 2, and pins 3 and 4 that will be taken advantage of. (then, when the switch closes, all of 1, 2, 3, and 4 are momentarily connected)

The "row wiring" needs to connect pins 1 and 2 of every key on the same row. This can be accomplished by bending pin 1 of one switch towards pin 2 of the next switch, then soldering them together. As you complete each row, use your multimeter in continuity test mode between the far left and right ends of the row to make sure everything is soldered.

When you proceed to the next row of keys, make sure to skip over a row of pins—We'll be soldering to pins 3 and 4 when it's time to do the "column wiring".

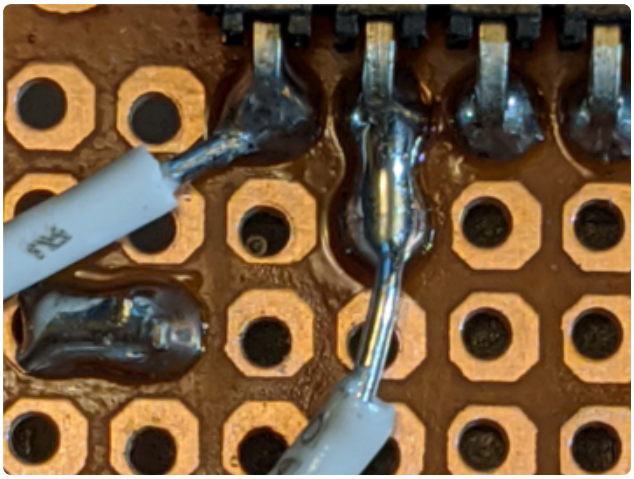


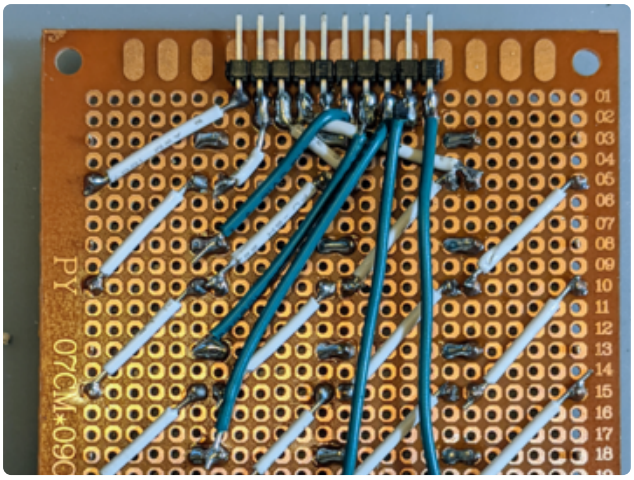


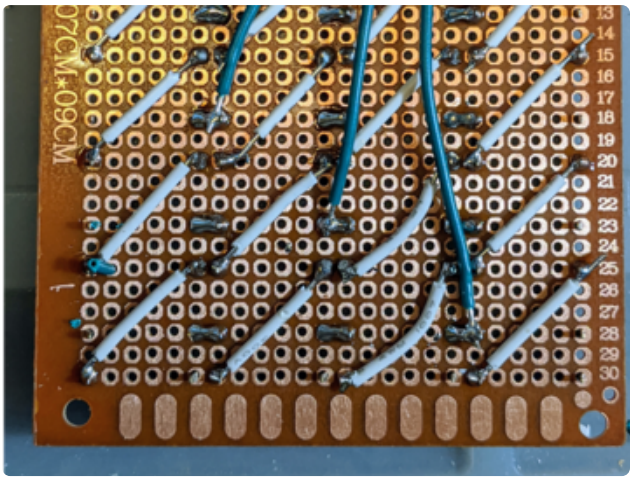


For the column wiring, I like to use a diagonal wire, from pin 3 of one button to pin 4 of this next one. This lets me place small pieces of wire, stripped at each end, and only solder one wire (not two) to each pin. The length of each wire in this case is about 2/3" or 16mm.

If, after soldering, the wire sticks beyond the end of the pin, trim it with flush cutters. As you complete each column, use your meter in continuity test mode to check that the connections are good from the top to bottom of the column.







Finally, we need to add a 10-pin header and connect each row and each column to it. No GND or VCC connection is needed.

Your calculator will look tidiest if you solder a right angle male header on the bottom (copper) side of the board, pointing out the top. Use the very topmost row of the perfboard.

Next, solder the first 6 positions with a connection from a "pin 3" or "pin 4" of each row.

Finally solder the other 4 positions with a connection from "pin 1" or "pin 2" of each row.

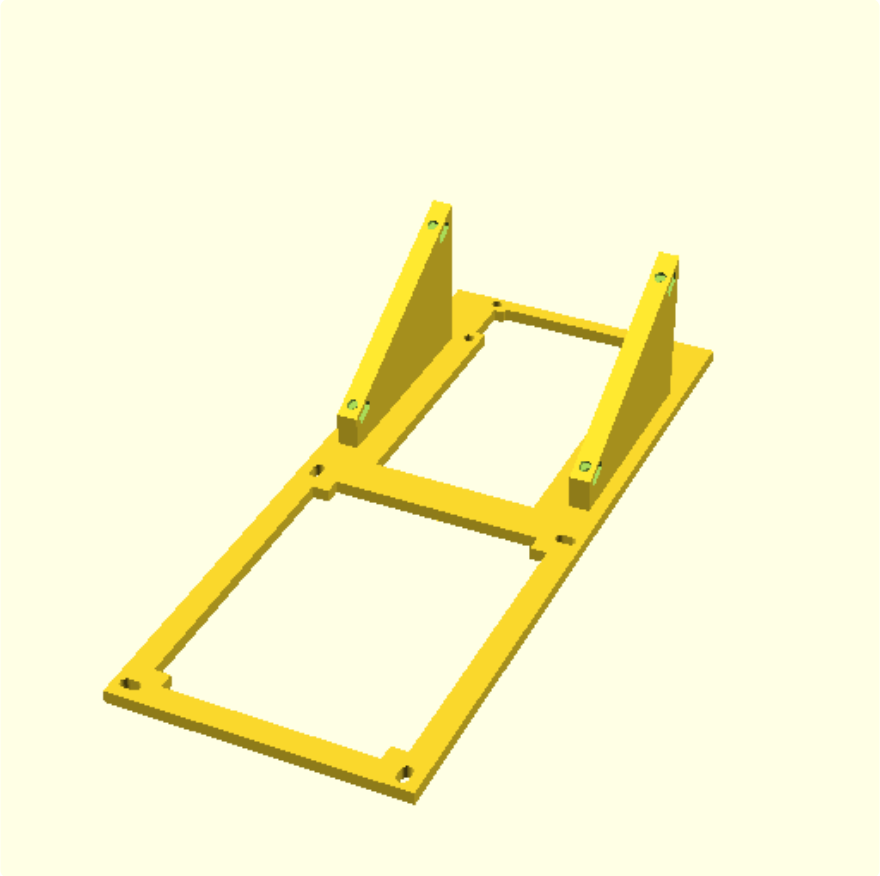
Phew! That was a lot of wiring! Now, for a final test get out your meter, make sure it's still set to continuity mode, and clip your probes to one row pin and one column pin. At first it will read open circuit, but when you press the button at that row and column it will beep for continuity. Check this for a few keys until you're satisfied the keyboard works properly. The microcontroller will do this rapidly for each combination of row and column when it "scans the matrix".

3D Printing, Wiring & Assembly

Download the zip file below and use the STL file inside for 3D printing. The source file, in "scad" format, is also included if you want to modify the design using OpenSCAD (https://adafru.it/OaC).

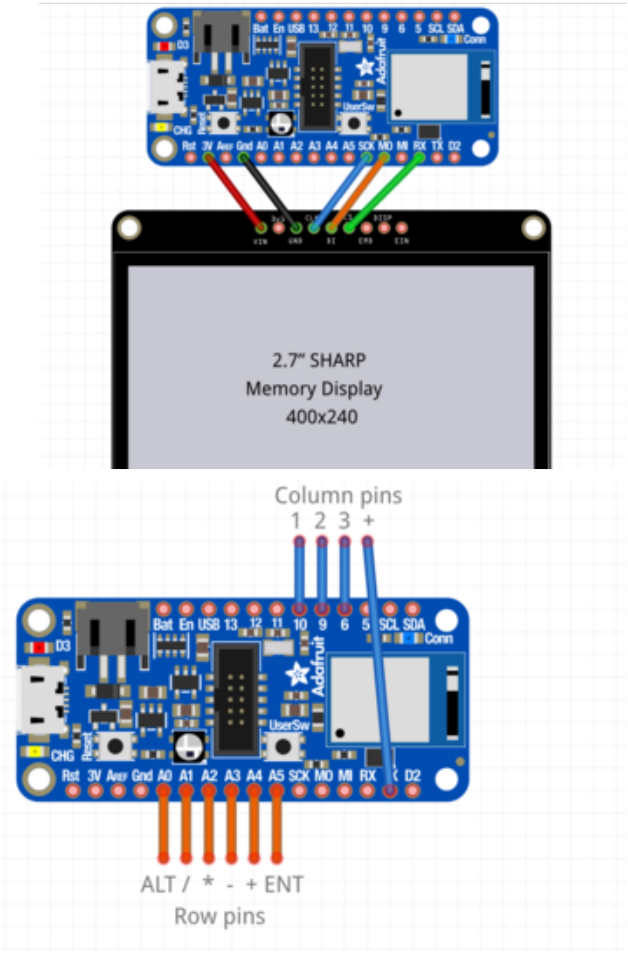
The print size is about 175 x 35 x 40mm.

The file is designed to print without supports using a layer height of 0.2mm or less.



calculon.zip

https://adafru.it/OaD



Solder male pins headers to the display and to your Feather, with the pins "down" as usual for breadboard use. Use 75mm jumper wires to wire the matrix and display to a CircuitPython Feather (such as the nRF52840 Feather) Keep the wires in bundles, especially the row and column bundles. Make the following connections:

Matrix "column" pins to **D10**, **D9**, **D6**, **TX**
Matrix "row" pins to **A0**, **A1**, **A2**, **A3**, **A4**, **A5**
Display pins to feather pins: **CLK** to **SCK**, **DI** to **MO**, **CS** to **RX**, **GND** to **GND**, **VIN** to **3V**

Assembly uses nylon M2.5 screws, nuts, and standoffs. Place the feather in back, pins up in the air. The best order for assembly I found is:

1. Loosely secure 4 short stand-offs in the keypad area. I used F-F standoffs and screws.
2. Place the keypad on top, adjust the stand-offs within the slots, and add screws from the top
3. Tighten the bottom screws until the keypad doesn't shift around anymore
4. Follow a similar procedure to secure the Feather
5. Insert nuts in the 4 slots, then 4 long M-F stand-offs into them.
6. Place the screen on top and secure with screws. Note the correct orientation of the screen.

It turns out not all nylon M2.5 nuts are created equal, so if yours don't fit in the little slots under the display, you'll have to get creative -- sand them down, hot glue the stand-off in place, modify the 3D print file, etc.

Installing the Code

Step 1 - Install CircuitPython

This guide requires CircuitPython be installed, click the button below to learn how to do that and install the latest version of CircuitPython.

Are you new to using CircuitPython? No worries, there is a full getting started guide here (https://adafru.it/cpy-welcome).

Adafruit suggests using the Mu editor to edit your code and have an interactive REPL in CircuitPython. You can learn about Mu and installation in this tutorial (https://adafru.it/ANO).

Install CircuitPython on the Feather nRF52840 Board

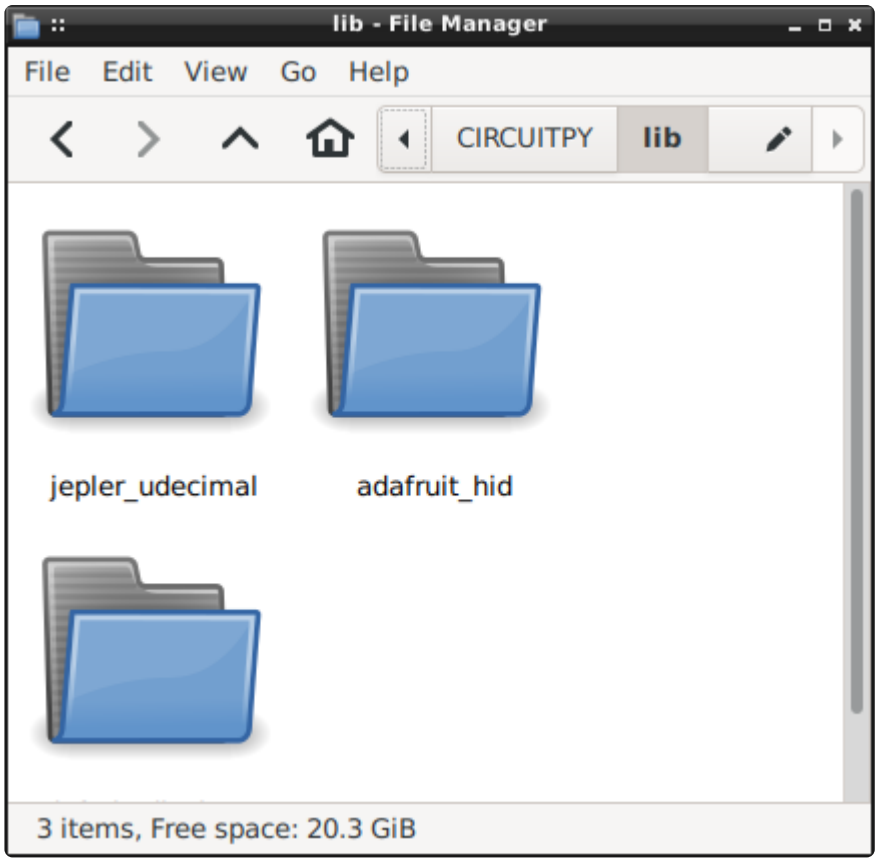
https://adafru.it/EwP

Step 2 - Install Libraries

First make sure you are running the latest version of Adafruit CircuitPython for the Feather nRF52840 (https://adafru.it/NCB). You'll also need to install several libraries on your Feather:

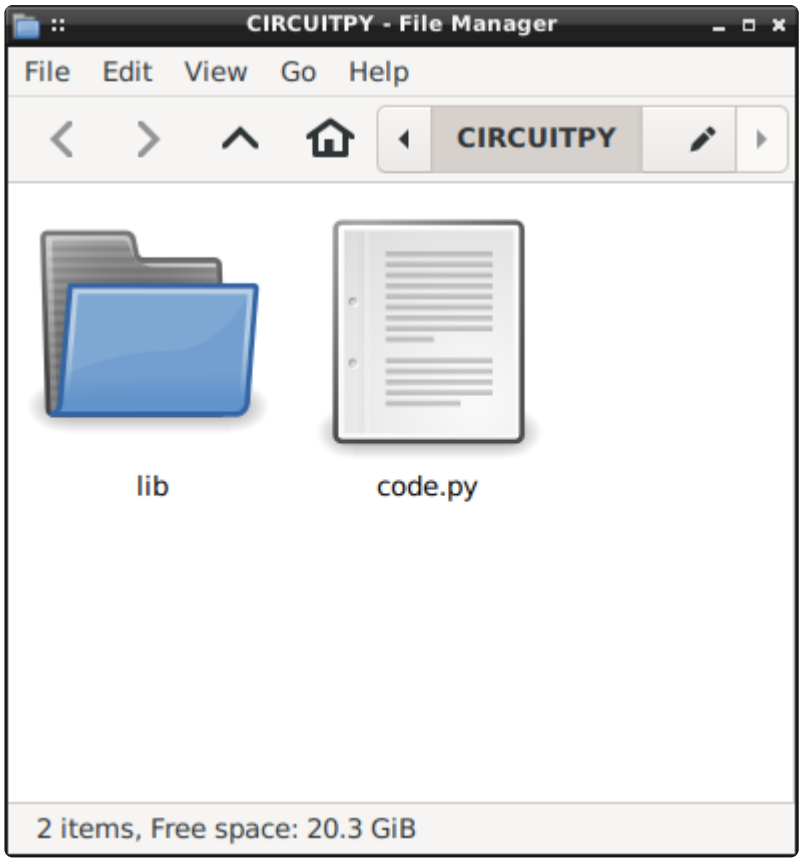
- **jepler_udecimal** (from the Community Bundle (https://adafru.it/OaE))
- **adafruit_hid**
- **adafruit_display_text**

Carefully follow the steps to find and install these libraries from Adafruit's CircuitPython Library Bundle (https://adafru.it/ENC). Our CircuitPython starter guide has a great page on how to install libraries from the bundle (https://adafru.it/ABU).



Step 3 - Install code

Use the "Project Zip" link to download the rest of the files needed for the calculator, then unzip it inside the **CIRCUITPY** drive.




```
# SPDX-FileCopyrightText: 2020 Jeff Epler for Adafruit Industries
#
# SPDX-License-Identifier: MIT

# pylint: disable=redefined-outer-name,no-self-use,broad-except,try-except-
raise,too-many-branches,too-many-statements,unused-import

import gc
import time

from adafruit_display_text.label import Label
from adafruit_hid.keyboard import Keyboard
from adafruit_hid.keyboard_layout_us import KeyboardLayoutUS
from jepler_udecimal import Decimal, getcontext, localcontext
import jepler_udecimal.utrig  # Needed for trig functions in Decimal
import board
import digitalio
import displayio
import framebufferio
import microcontroller
import sharpdisplay
import terminalio

try:
    import usb_hid
except ImportError:
    usb_hid = None

# Initialize the display, cleaning up after a display from the previous
# run if necessary
displayio.release_displays()
framebuffer = sharpdisplay.SharpMemoryFramebuffer(board.SPI(), board.RX, 400, 240)
display = framebufferio.FramebufferDisplay(framebuffer, auto_refresh=False)

def extraprec(add=8, num=0, den=1):
    def inner(fn):
```

```
        def wrapper(*args, **kw):
            with localcontext() as ctx:
                ctx.prec = ctx.prec + add + (ctx.prec * num + den - 1) // den
                result = fn(*args, **kw)
            return +result
        return wrapper
    return inner

class AngleConvert:
    def __init__(self):
        self.state = 0

    def next_state(self):
        self.state = (self.state + 1) % 3

    def __str__(self):
        return "DRG"[self.state]

    @property
    def factor(self):
        return [360, None, 400][self.state]

    def from_user(self, x):
        factor = self.factor
        if factor is None:
            return x
        x = x.remainder_near(factor)
        pi_4 = Decimal("1.0").atan()
        return x * pi_4 * 8 / factor

    def to_user(self, x):
        factor = self.factor
        if factor is None:
            return x
        pi_4 = Decimal("1.0").atan()
        return x * factor / pi_4 / 8

    @extraprec(num=1)
    def cos(self, x):
        return self.from_user(x).cos()

    @extraprec(num=1)
    def sin(self, x):
        return self.from_user(x).sin()

    @extraprec(num=1)
    def tan(self, x):
        return self.from_user(x).tan()

    @extraprec(num=1)
    def acos(self, x):
        return self.to_user(x.acos())

    @extraprec(num=1)
    def asin(self, x):
        return self.to_user(x.asin())

    @extraprec(num=1)
    def atan(self, x):
        return self.to_user(x.atan())

getcontext().prec = 14
getcontext().Emax = 99
getcontext().Emin = -99

def get_pin(x):
    if isinstance(x, microcontroller.Pin):
        return digitalio.DigitalInOut(x)
    return x
```

```
class MatrixKeypadBase:
    def __init__(self, row_pins, col_pins):
        self.row_pins = [get_pin(p) for p in row_pins]
        self.col_pins = [get_pin(p) for p in col_pins]
        self.old_state = set()
        self.state = set()

        for r in self.row_pins:
            r.switch_to_input(digitalio.Pull.UP)
        for c in self.col_pins:
            c.switch_to_output(False)

    def scan(self):
        self.old_state = self.state
        state = set()
        for c, cp in enumerate(self.col_pins):
            cp.switch_to_output(False)
            for r, rp in enumerate(self.row_pins):
                if not rp.value:
                    state.add((r, c))
            cp.switch_to_input()
        self.state = state
        return state

    def rising(self):
        old_state = self.old_state
        new_state = self.state

        return new_state - old_state

class LayerSelect:
    def __init__(self, idx=1, next_layer=None):
        self.idx = idx
        self.next_layer = next_layer or self

LL0 = LayerSelect(0)
LL1 = LayerSelect(1)
LS1 = LayerSelect(1, LL0)

class MatrixKeypad:
    def __init__(self, row_pins, col_pins, layers):
        self.base = MatrixKeypadBase(row_pins, col_pins)
        self.layers = layers
        self.layer = LL0
        self.pending = []

    def getch(self):
        if not self.pending:
            self.base.scan()
            for r, c in self.base.rising():
                op = self.layers[self.layer.idx][r][c]
                if isinstance(op, LayerSelect):
                    self.layer = op
                else:
                    self.pending.extend(op)
                    self.layer = self.layer.next_layer

        if self.pending:
            return self.pending.pop(0)

        return None

col_pins = (board.D10, board.D9, board.D6, board.TX)
row_pins = (board.A0, board.A1, board.A2, board.A3, board.A4, board.A5)

BS = '\x7f'
CR = '\n'
```

```
layers = (
    (
        ('^', 'l', 'r', LS1),
        ('s', 'c', 't', '/'),
        ('7', '8', '9', '*'),
        ('4', '5', '6', '-'),
        ('1', '2', '3', '+'),
        ('0', '.',  BS,  CR)
    ),

    (
        ('v', 'L', 'R', LL0),
        ('S', 'C', 'T', 'N'),
        ( '',  '',  '',  ''),
        ( '',  '',  '', 'n'),
        ( '',  '',  '',  ''),
        ('=', '@',  BS, '~')
    ),
)


class Impl:
    def __init__(self):
        # incoming keypad
        self.keypad = MatrixKeypad(row_pins, col_pins, layers)

        # outgoing keypresses
        self.keyboard = None
        self.keyboard_layout = None

        g = displayio.Group()

        self.labels = labels = []
        labels.append(Label(terminalio.FONT, scale=2, color=0))
        labels.append(Label(terminalio.FONT, scale=3, color=0))
        labels.append(Label(terminalio.FONT, scale=3, color=0))
        labels.append(Label(terminalio.FONT, scale=3, color=0))
        labels.append(Label(terminalio.FONT, scale=3, color=0))
        labels.append(Label(terminalio.FONT, scale=3, color=0))

        for li in labels:
            g.append(li)

        bitmap = displayio.Bitmap((display.width + 126)//127, (display.height +
126)//127, 1)
        palette = displayio.Palette(1)
        palette[0] = 0xffffff

        tile_grid = displayio.TileGrid(bitmap, pixel_shader=palette)
        bg = displayio.Group(scale=127)
        bg.append(tile_grid)

        g.insert(0, bg)

        display.root_group = g

    def getch(self):
        while True:
            time.sleep(.02)
            c = self.keypad.getch()
            if c is not None:
                return c

    def setline(self, i, text):
        li = self.labels[i]
        text = text[:31] or " "
        if text == li.text:
            return
        li.text = text
```

```
        li.anchor_point = (0,0)
        li.anchored_position = (1, max(1, 41 * i - 7) + 6)

    def refresh(self):
        pass

    def paste(self, text):
        if self.keyboard is None:
            if usb_hid:
                self.keyboard = Keyboard(usb_hid.devices)
                self.keyboard_layout = KeyboardLayoutUS(self.keyboard)
            else:
                return

        if self.keyboard_layout is None:
            raise ValueError("USB HID not available")
        text = str(text)
        self.keyboard_layout.write(text)
        raise RuntimeError("Pasted")

    def start_redraw(self):
        display.auto_refresh = False

    def end_redraw(self):
        display.auto_refresh = True

    def end(self):
        pass
impl = Impl()

stack = []
entry = []

def do_op(arity, fun):
    if arity > len(stack):
        return "underflow"
    res = fun(*stack[-arity:][::-1])
    del stack[-arity:]
    if isinstance(res, list):
        stack.extend(res)
    elif res is not None:
        stack.append(res)
    return None
angleconvert = AngleConvert()

def roll():
    stack[:] = stack[1:] + stack[:1]

def rroll():
    stack[:] = stack[-1:] + stack[:-1]

def swap():
    stack[-2:] = [stack[-1], stack[-2]]

ops = {
    '\'': (1, lambda x: -x),
    '\\': (2, lambda x, y: x/y),
    '#': (2, lambda x, y: y**(1/x)),
    '*': (2, lambda x, y: y*x),
    '+': (2, lambda x, y: y+x),
    '-': (2, lambda x, y: y-x),
    '/': (2, lambda x, y: y/x),
    '^': (2, lambda x, y: y**x),
    'v': (2, lambda x, y: y**(1/x)),
    '_': (2, lambda x, y: x-y),
    '@': angleconvert.next_state,
    'C': (1, angleconvert.acos),
    'c': (1, angleconvert.cos),
    'L': (1, Decimal.exp),
```

```
    'l': (1, Decimal.ln),
    'q': (1, lambda x: x**.5),
    'r': roll,
    'R': rroll,
    'S': (1, angleconvert.asin),
    's': (1, angleconvert.sin),
    '~': swap,
    'T': (1, angleconvert.atan),
    't': (1, angleconvert.tan),
    'n': (1, lambda x: -x),
    'N': (1, lambda x: 1/x),
    '=': (1, impl.paste)
}

def pstack(msg):
    impl.setline(0, f'[{angleconvert}] {msg}')

    for i, reg in enumerate("TZYX"):
        if len(stack) > 3-i:
            val = stack[-4+i]
        else:
            val = ""
        impl.setline(1+i, f"{reg} {val}")

def loop():
    impl.start_redraw()
    pstack(f'{gc.mem_free()} RPN bytes free')
    impl.setline(5, "> " + "".join(entry) + "_")
    impl.refresh()
    impl.end_redraw()

    while True:
        do_pstack = False
        do_pentry = False
        message = ''


        c = impl.getch()
        if c in '\x7f\x08':
            if entry:
                entry.pop()
                do_pentry = True
            elif stack:
                stack.pop()
                do_pstack = True
        if c == '\x1b':
            del entry[:]
            do_pentry = True
        elif c in '0123456789.eE':
            if c == '.' and '.' in entry:
                c = 'e'
            entry.append(c)
            do_pentry = True
        elif c == '\x04':
            break
        elif c in ' \n':
            if entry:
                try:
                    stack.append(Decimal("".join(entry)))
                except Exception as e:
                    message = str(e)
                del entry[:]
            elif c == '\n' and stack:
                stack.append(stack[-1])
            do_pstack = True
        elif c in ops:
            if entry:
                try:
                    stack.append(Decimal("".join(entry)))
```

```
                except Exception as e:
                    message = str(e)
                del entry[:]
            op = ops.get(c)
            try:
                if callable(op):
                    message = op() or ''
                else:
                    message = do_op(*op) or ''
            except (KeyboardInterrupt, SystemExit):
                raise
            except Exception as e:
                message = str(e)
            do_pstack = True

        impl.start_redraw()

        if do_pstack:
            pstack(message)
            do_pentry = True

        if do_pentry:
            impl.setline(5, "> " + "".join(entry) + "_")

        if do_pentry or do_pstack:
            impl.refresh()

        impl.end_redraw()

try:
    loop()
finally:
    impl.end()
```

Using your calculator

The stack

There is an unlimited stack of numbers. The last 4 entries are displayed as "T", "Z", "X" and "Y". The number currently being entered is shown on the line marked ">".

Entering numbers

Thanks to the jepler_udecimal library, the calculator uses 14-digit numbers and allows exponents from -99 to +99.

Enter a simple number by pressing digit keys followed by ENT, such as "1234". This places the number at the last entry of the stack and moves the other entries deeper in the stack.

The "decimal point" key doubles as the exponent key, so to enter the number "6.02e23" enter "6.02.23". For "1E3" type "1..3".

Performing operations

To perform an operation, enter the required operands and then press the operation key. If you were in the middle of entering a number when you press an operation key, the number is automatically Entered.

For example, to add "1" and "2", you can type: "1 ENT 2 +". "1" and "2" are removed, and "3" is placed on the stack.

Alternate Function

Some keys have two functions. To access the alternate function, press and release ALT, then press the other key.

Pasting to PC

When you press Paste (ALT 0), the "X" number is entered via USB HID on the attached computer. The number remains on the stack.

Keypad Reference

Here are all the functions you can access within the calculator. The left hand side is the normal key, and the right hand side is the alternate function.

Y^X	ln X	roll up	ALT		$Y^{1/X}$	e^X	roll down	ALT
sin X	cos X	tan X	Y/X		$sin^{-1}X$	$cos^{-1}X$	$tan^{-1}X$	1/X
7	8	9	Y*X					
4	5	6	Y-X					X-Y
1	2	3	Y+X					
0	. / E	BS / DROP	ENT / DUP		PASTE	DRG	BS	SWAP

Code Highlights and Customization

The code is long enough (around 400 lines) that a line by line explanation would be too verbose. Here are some high level notes to give you an overview:

```
class AngleConvert:
    ...
```

The AngleConvert class helps implement the "degrees / radians / gradians" mode common on scientific calculators. It wraps each of the 6 trig functions, converting to/from radians as necessary. To improve the precision of the result, these steps are carried out with extra digits of precision using the `extraprec` decorator function.

```
getcontext().prec = 14
getcontext().Emax = 99
getcontext().Emin = -99
```

Sets the default precision (number of decimal places), minimum and maximum exponents. These values were chosen so that any number should fit on the screen without being cut off, but any of these values can be increased or decreased if desired.

```
class MatrixKeypadBase:
    ...

class MatrixKeypad:
    ...

class LayerSelect:
    ...

...
layers = (
    (
        ('^', 'l', 'r', LS1),
        ...
    ),
    ...
)
```

These classes implement a matrix keyboard, including layer selection. The "layers" tuple has "layer 0" (normal layer) followed by "layer 1" (alternate layer). Except for the special values `LS1` (layer shift 1) and `LL0` (layer lock 0), each one specifies a (possibly empty) string that is handled later by the the dictionary of `ops` or by the main `loop`.

```
class Impl:
    ...
```

The "Impl" (implementation) class has the details of how to interact with the keyboard matrix and update the display. The calculator program originally ran on a PC in a terminal window, and this class is a vestige of trying to support for both CircuitPython and standard Python3 within a single program.

```
def do_op(arity, fun):
    ...

ops = {
    '\'': (1, lambda x: -x),
    '\\': (2, lambda x, y: x/y),  # keypad: SHIFT+/
    ...
    '@': angleconvert.next_state,
    ...
}
```

`ops` is a dictionary where the **key** is a character produced by the key matrix and the **value** is either a **callable** or a tuple of **(arity, callable)**.

When an operation is a **callable**, it has to manage the stack itself. When it's a tuple, then the first number in the tuple (called the **arity**) specifies how many arguments the function expects. That number of arguments are passed as arguments to the function. If the function succeeds, the arguments are removed from the stack. Then, the return value, if it's not **None**, is put back on the stack.

```
def pstack(msg):
    ...
```

The pstack function updates the screen.

```
def loop():
    ...
```

The loop function runs continuously, waiting for key to be entered. Some keys are processed specially (like the digits, "./E", backspace, and enter/dup), other keys are handled by looking up the operation in `ops`.

Adding new functions

To add a new function, you will need several things:

- The implementation of the mathematical function. This can be a function that you write, but for this example we will use the existing function `Decimal.log10`
- The physical key location you choose for the function. Let's choose the ALT function of the key that is normally "+"

- The character you use to represent it. Let's choose "O" (capital Oscar), perhaps standing for the "o" of **log10** but mostly because it is not used yet.
- The arity of the function. **log10** takes a single number (x), so its arity is 1

With all these pieces of information, we can start to modify the code. First, we have to modify the keymap, called **layers**:

```
layers = (
    (
        ('^', 'l', 'r', LS1),
        ('s', 'c', 't', '/'),
        ('7', '8', '9', '*'),
        ('4', '5', '6', '-'),
        ('1', '2', '3', '+'),
        ('0', '.',  BS,  CR)
    ),

    (
        ('v', 'L', 'R', LL0),
        ('S', 'C', 'T', 'N'),
        ( '',  '',  '',  ''),
        ( '',  '',  '', 'n'),
        ( '',  '',  '', 'O'), # modify this line, changing '' to 'O'
        ('=', '@',  BS, '~')
    ),
)
```

Next, add `'O': Decimal.log10,` to `ops`:

```
ops = {
    'O': (1, Decimal.log10),   # Add this line
    '\'': (1, lambda x: -x),
    ...                        # Keep the other lines as-is
}
```

When you save the file, CircuitPython will automatically restart. Type "1000 ALT +" and you should see the result "3", because **1000=10^3**.

Here are some other ideas about functions to add to the calculator:

- Unit conversions like inches to millimeters
- Constants like π or e. π can be computed as `Decimal('1').atan() * 4`, and e can be computed as `Decimal('1').exp()`
- Engineering functions, statistical functions, etc

Other ideas for customization and improvement

- The Feather nRF52840 can also act as a BLE (bluetooth) keyboard. Convert the calculator to run from a rechargeable battery and make it paste over Bluetooth instead of or in addition to USB.

- Using the jepler_udecimal library, implement a standard "infix" calculator instead.
- Add the ability to enter a formula from the keypad and create a CircuitPython graphing calculator