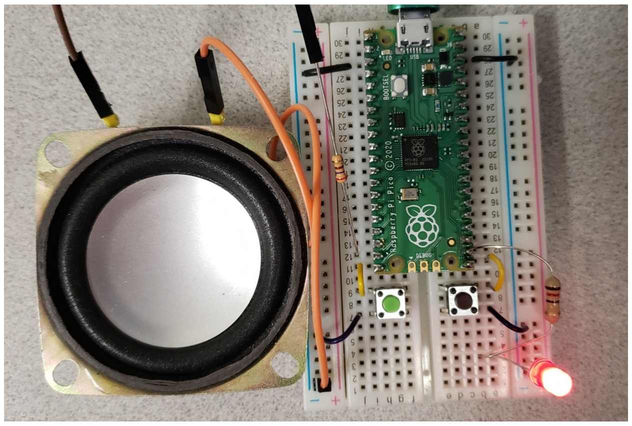
Breadboards, Raspberry Pi Pico, & Python



Sample breadboard circuit with a Raspberry Pi Pico

Introduction

The goal of ECE 401 is that by the end of the semester, each student has a custom-built, working PCB he/she can demonstrate and explain. To get there, you usually

- Design your circuit on paper,
- Test your design in CircuitLab,
- Build your design on a breadboard,
- Once your breadboard is working, transfer your design to a PCB,
- Then solder, test, and demonstrate your PCB

Breadboards are an important step in this process. With a breadboard, you can verify that your paper design is in fact correct. More importantly, breadboards allow you to adjust your design fairly easily if needed. Much easier (and faster and less expensively) than trying to modify and correct a PCB.

In this lecture, we'll look at do's and don'ts of breadboarding a circuit. We'll also look at setting up a Raspberry Pi Pico on a breadboard.

Note: Many but not all ECE 401 projects use a microcontroller. Whether you do or don't use a microcontroller in your design is up to you. However, microcontrollers are just darn useful. For just the price of $4, you can include a Raspberry Pi-Pico into your design (paid by the ECE department - no cost to the student). By incorporating a microcontroller, you can simplify the hardware design of many circuits as well as a great deal of flexibility to your design.

The Raspberry Pi-Pico along with Python is used in this class because

- Python is a very friendly language. If you're familiar with Matlab, you basically know Python.
- The Raspberry Pi Pico is a very powerful microcontroller. With it, you have access to
    - Binary inputs and outputs,
    - Analog inputs with three 12-bit A/D inputs,
    - Analog outputs with up to 30 PWM channels,
    - Sensors and actuators which use I2C, SPI, and SCI interfaces,
    - Bluetooth and WiFi,
    - and many more features

Many projects in Senior Design II and III use the Raspberry Pi Pico - so getting familiar with it in Senior Design I puts you further ahead when you get to Design II and III.

Breadboards

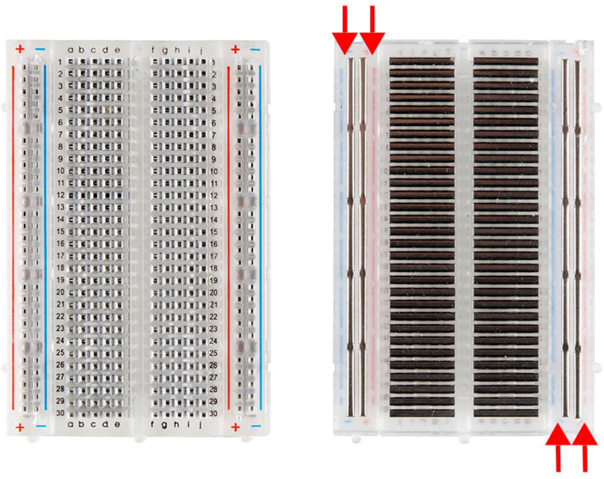
Once you have your circuit

- Designed on paper, and
- Tested in simulation (CircuitLab)

you're ready to test your design in hardware. Breadboards are an easy way to build your circuit and test your design. They're also easy to modify and change: components can be easily added and removed from a breadboard.

Most of the breadboard used in ECE 401 are 400 tie breadboards. These have

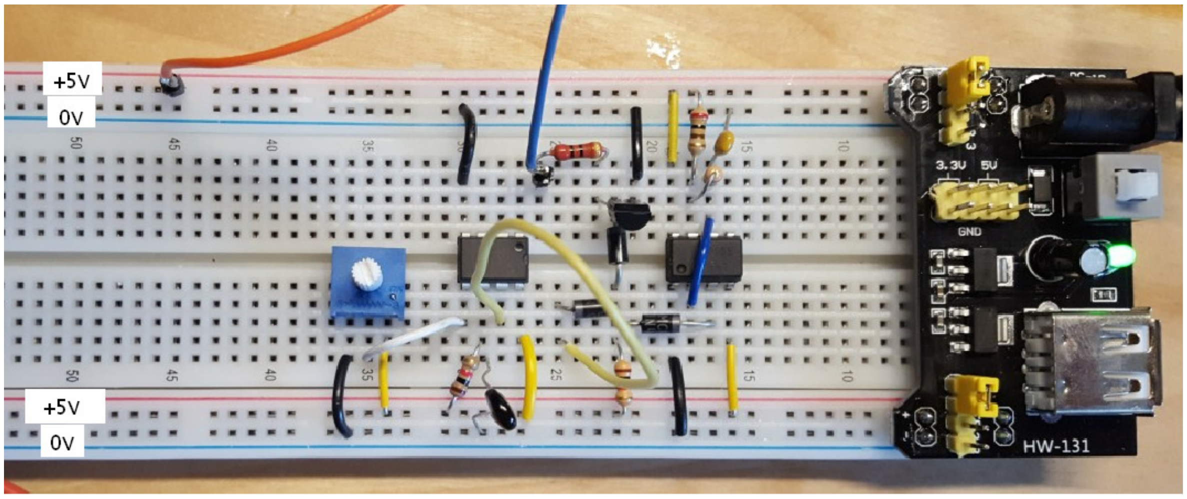
- Four edge connectors that are shorted along the length of the breadboard (two left side, two right side). These are usually used for power and ground.
- Two sets of connectors in the middle (one left and one right of the center bar). These are shorted horizontally.
- Across the middle is an insulator - this separates the middle connectors by 300 mils: the width of a typical IC,



Breadboard Top (left) and back side (right)

For example, the following breadboard circuit uses

- The red trace along the top and bottom as +5V
- The blue trace along the top and bottom as 0V
- Two IC's go across the middle divider.
- The four pins above and below each IC then allow you to connect to that pin



Typical Use of a Breadboard.
Top and bottom rails are used for power and ground
ICs go across the center divide
Each column of pins above and below the IC connect to that IC's pin in that column

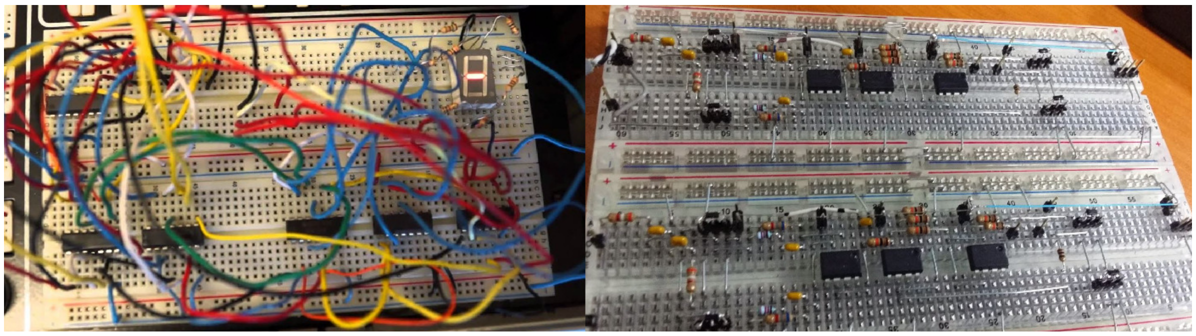
Tricks of Breadboarding

1) Keep Your Circuit Neat

- Use short wires
- Use short component leads
- Organize your breadboard into sections

Keeping your wires short

- Reduces the noise picked up by your wires
- Reduces the chance of a wire falling out
- Helps you see the wiring in your board
- Helps when you need to modify your breadboard circuit.



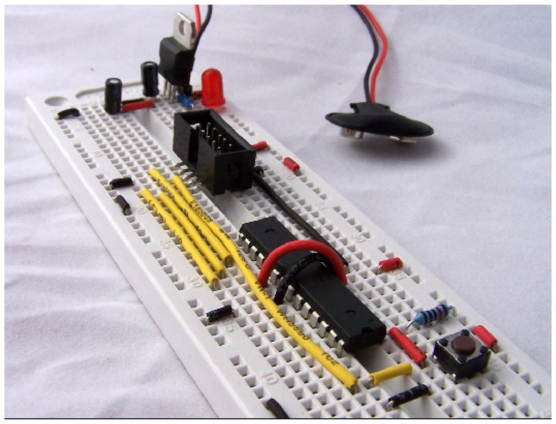
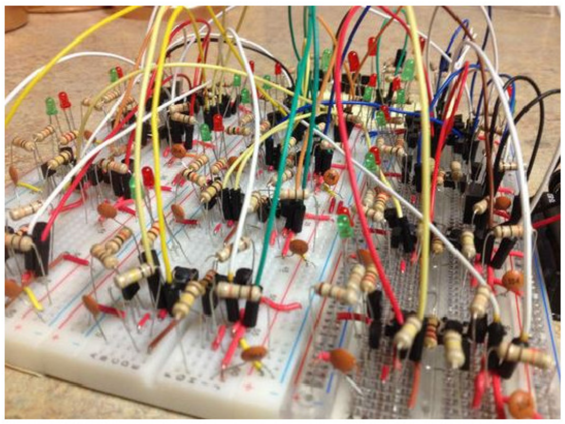
Do use short wires. Keep your breadboard neat and organized.
The circuit on the left will be very noisy and very hard to debug

2. Color Code your Wires

- Use red wires for +5V
- Use black wires for ground
- Use different colors for different types of signals.

By color coding your wires,

- You can quickly spot if a chip is missing power and/or ground.
- You can quickly see if a signal wire is missing between two ICs

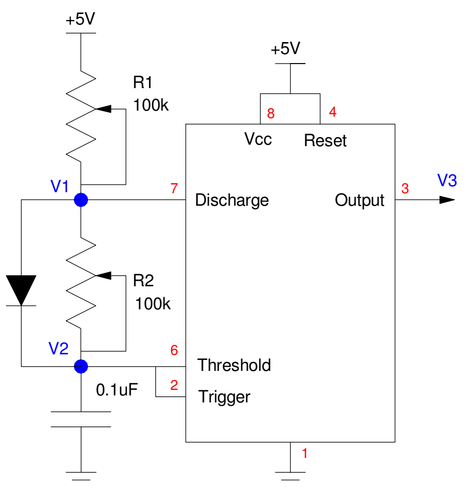
Do color code your wires. It makes it much easier to spot missing wires or wires going to the wrong spot.

Better chance of seeing which wire is in the wrong spot

3. Use Potentiometers (2 max)

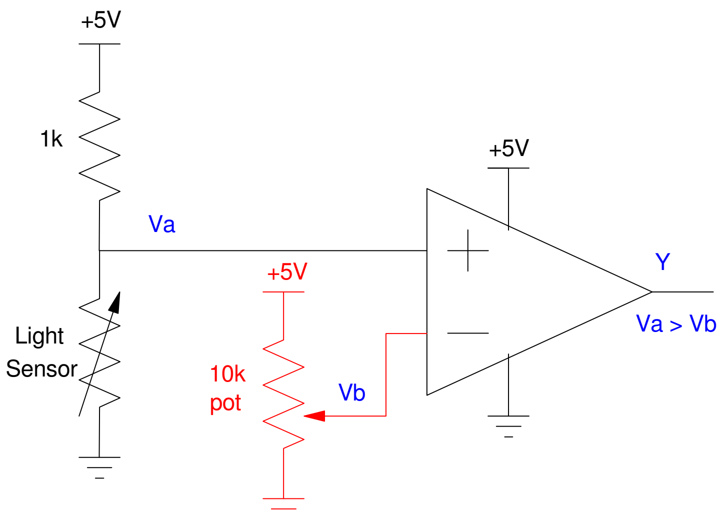
Often times, you're going to need to adjust a resistor in order to make a circuit work. For example, when using a 555 timer, the resistors R1 and R2 set the on and off times. If you replace these with potentiometers, you can adjust them on your breadboard to tune your circuit and get the on/off times within your tolerances

For example, if R1 and R2 are calculated to be 72k and 52k Ohms respectively, you could use two 100k potentiometers connected as follows:



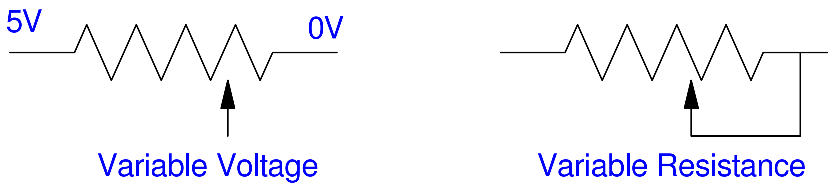
By replacing R1 and R2 with potentiometers, you can adjust each over the range of 0k to 100k

Potentiometers can also be used to provide an adjustable voltage. For example, the following circuit uses a comparitor for a night light. The potentiometer allows you to adjust the voltage at Vb from 0V to 5V - allowing you to adjust the light level where the night-light turns on and off.



Pots allow you to adjust a voltage - allowing you to adjust the voltage (or light level) the comparitor switches at

The basic circuit for a pot when used as a variable voltage or resistance is:



Two common configurations for potentiometers

Potentiometers are a little expensive, however:

- Resistor = $0.02
- Pot: $1.55

So, for ECE 401, please limit your circuit to using two potentiometers or less.

4. Breadboards & Test Points

Some things to think about when using a breadboard are:

- How do you test your circuit?
- What signal do you look at?
- What should the signals look like?
- What procedures do you use?

Note what signals you look at and record what you read. This affects your upcoming PCB layout:

- These same signals should be measured in simulation and on your PCB
- Test points should be added to your PCB so that you have access to these signals.

5. Keep Your Breadboard

When done testing your breadboard, keep it together, intact (i.e. don't cannibalize it for parts). If your PCB doesn't work properly, your (working) breadboard circuit will be helpful in debugging what part of your PCB works (and has similar signals), and which part does not work.

This means you'll need two of every part in ECE 401

- One for your breadboard circuit, and
- One for your PCB.

That's OK.

Test Points

Once you build your breadboard, you need to test it to make sure it works. When you do so, pay attention to what signals you're looking at. When you build your PCB, you need to make sure you have access to these signals (i.e. add test points to your PCB).

Also also, in OneNote, record

- The conditions of the test,
- What you are measuring (show on the schematic), and
- What the output was (usually a multimeter or oscilloscope reading)

This will be needed later on when you're testing your PCB.

Each project must include at least six test points. These include

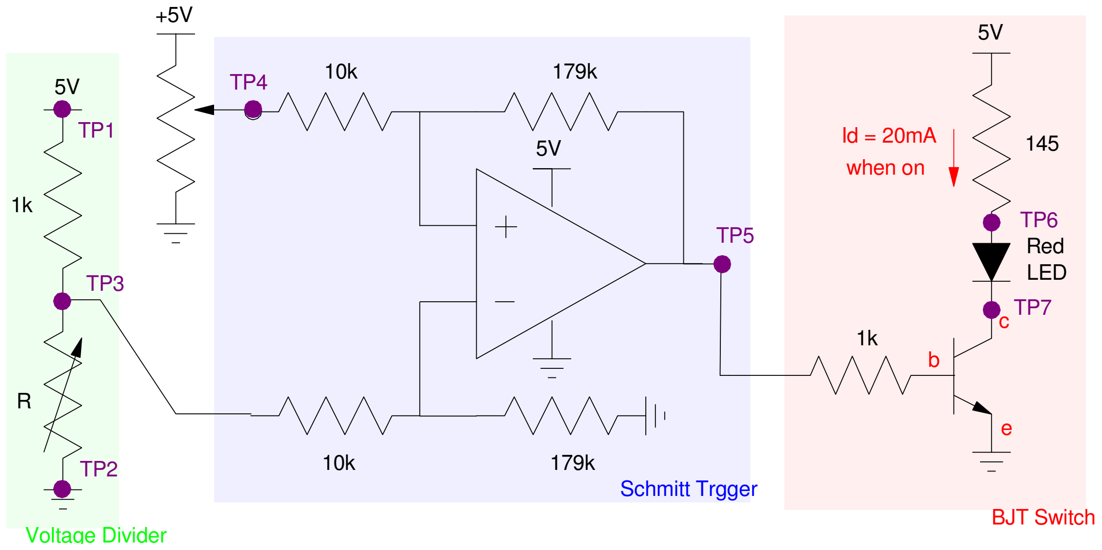
- 9V power
- 5V power
- Ground (really important - everything is measured relative to ground)
- The collector of each transistor (needed to check if the BJT is saturated)

Other signals are OK to include as well: this is just the minimum.

Just pay attention to what you are measuring when you verify your breadboard is working.

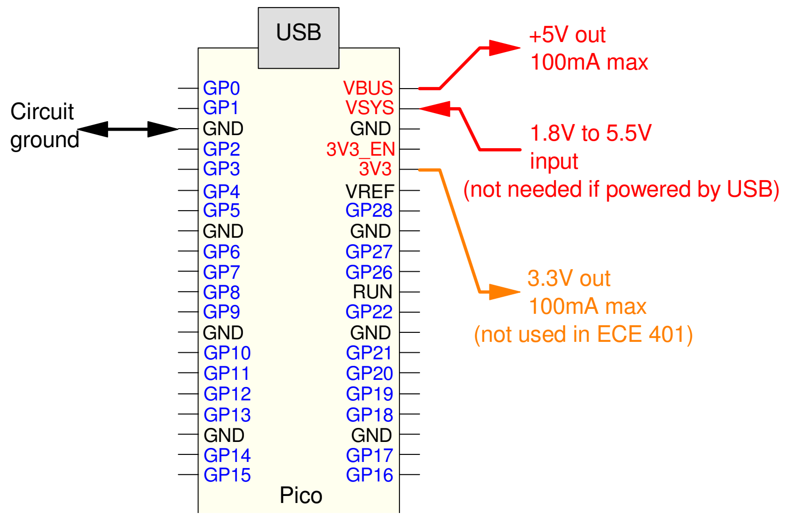
- These measurements should be recorded in your OneNote document,
- Your PCB should include test points at each of these signals as well.

For example, with the Schmitt Trigger from before, eight test points would be useful when testing and debugging the circuit:



One possible set of test points for a Schmitt Trigger

Raspberry Pi-Pico



In Senior Design I, many of the projects can be built just using digital logic and 555 timers. They could also be built *using* a microcontroller.

Microcontrollers are just a tool: if the tool helps you do your job, use it. If not, don't use it. If you don't use a microcontroller, you don't need to worry about

- Designing hardware around the microcontroller,
- Having to write and debug code, and
- How to download that code.

If you are willing to learn how to do this, however, microcontrollers can give you a great deal of flexibility in your design.

In this lecture, we going to cover

- Hardware: How to wire up a Raspberry Pi-Pico so that you can make a light blink
- Software: Programming a Pi-Pico using Thonny and Python

Hardware:

The minimum connections you need to wire up your Pico chip to a breadboard depend upon how you want to supply power:

Power from the USB Cord

- Connect GND to circuit ground
- Connect VBUS to the breadboard's +5V rail for the rest of your circuit (max current = 100mA)

Power from 9V battery

- Step the voltage down to 5V using a DC-to-DC converter
- Connect 5V to VSYS to power the Pi-Pico
  - VSYS accepts any voltage in the range of (1.8V, 5.5V)
- Conenct GND to circuit ground

Once your Pico has power and ground, you can read and write binary signals from each of the GPIO pins.

Binary Outputs: Each general purpose I/O pin can be a binary input or binary output. When used as a binary output, each pin can source or sink up to 12mA where

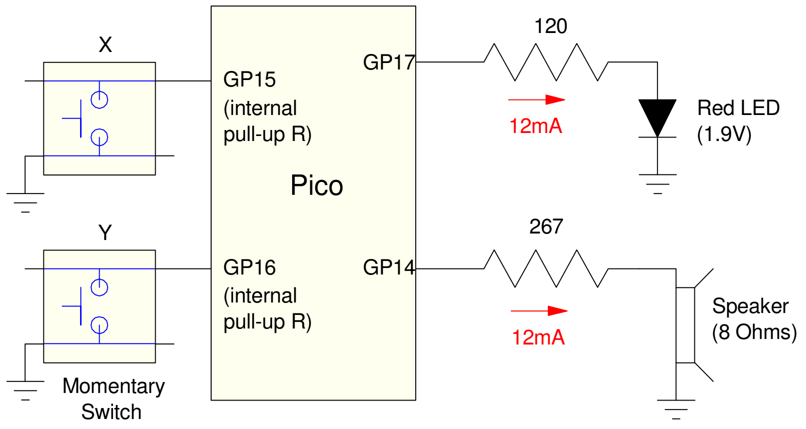
- 0V is logic 0
- 3.3V is logic 1

Binary Inputs: When used as an input, each I/O pin recognizes

- (0.0V - 0.8V) as logic 0
- (2.0V to 3.5V) as logic 1

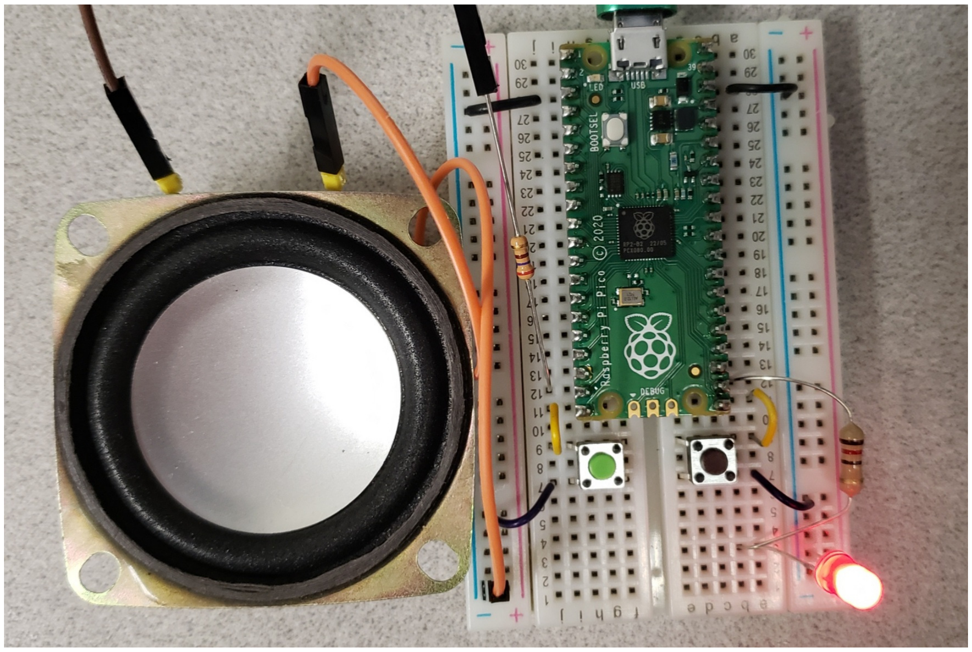
Note: **Do not connect 5.0V to the Pi-Pico's input pins. This may damage the Pi-Pico.**

Each input pin can also be connected to an internal pull-up or pull-down resistor. (more on this later). For example, the schematic for connecting two buttons (binary inputs) and an LED (binary output) to a Pico would be as follows. The resistor to the LED limits the current to 12mA (more on this when we get to LEDs)



Sample circuit where a Pico reads two push buttons and drives and LED and a speaker

The resulting breadboard would look like this:



Corresponding breadboard circuit with a Raspberry Pi Pico connected to two buttons, an LED, and a speaker

Software: Thonny and MicroPython

Several programming languages are available for a Pi-Pico, including

- Assembler
- C
- Python (MicroPython for a Pi-Pico)

among others. In this lecture, we focus on MicroPython.

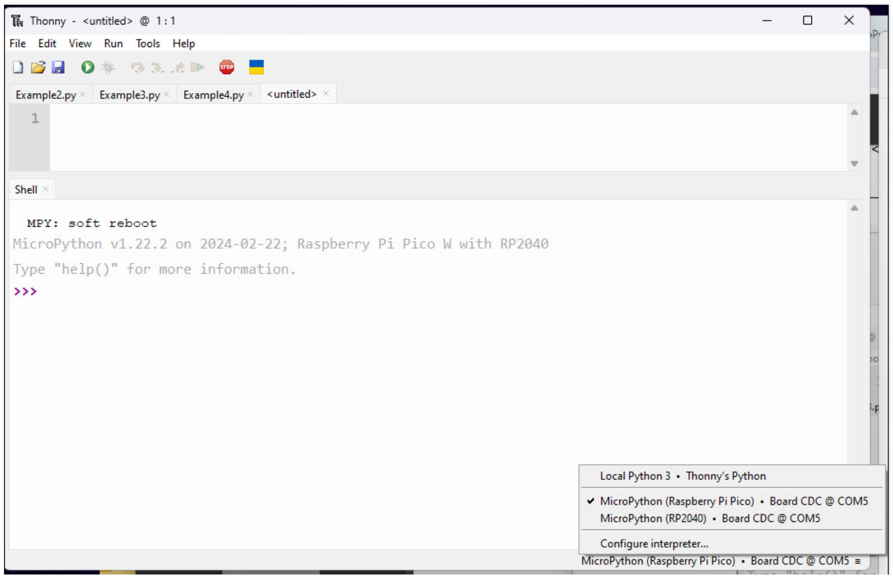
MicroPython is version of Python with reduced functionality so that it fits on a microcontroller like a Pi-Pico. Python is very similar to Matlab:

- Both are interpretive languages: Each line of code is executed with the results available to the user as the program executes.
- Both use similar syntax: Code that works in Matlab mostly works in Python
- Both use a similar console: the user has a command window and a script window

If you start Thonny (the Python compiled used in this class), you'll see the following:

- The top of the screen has icons for what you want to do including
  - File New / Open / Save
  - Run (run the program in the script window)
  - Stop (stop the program that's currently executing - clear memory)
  - Donate to Ukraine
- The top window is the script window: These are programs you can execute

- The bottom window is the shell window (command window in Matlab-speak).
- The lower-right corner is the Pi-Pico you're connected to



```
 MPY: soft reboot
MicroPython v1.22.2 on 2024-02-22; Raspberry Pi Pico W with RP2040
Type "help()" for more information.
>>>
```

Local Python 3 • Thonny's Python
✓ MicroPython (Raspberry Pi Pico) • Board CDC @ COM5
MicroPython (RP2040) • Board CDC @ COM5
Configure interpreter...
MicroPython (Raspberry Pi Pico) • Board CDC @ COM5

Python is similar to Matlab: you can type commands directly in the shell window. For example, you can use Python like a calculator:

Shell

```
>>>  a = 2
>>>  b = 3
>>>  c = 2*a + 3*b + 4
>>>  print c
    17
```

You can also place this code in the script window and execute by pressing the Run button (green arrow)

```
a = 2
b = 3
c = 2*a + 3*b + 4
print(c)
```

Shell

```
>>>
    17
```

Python is also similar to Matlab in that you don't have to declare variables at the start of your program. Instead, you can create them on the fly as in the above examples. In addition, Python will automatically change the variable type on the fly.

For example, a and b are both integers in the above program. If c is the radio of a/b, c automatically becomes a floating point number.]

Binary I/O with Python

Python is a little different than Matlab. For one thing, to use functions in a library, you have to use the *import* command. This makes that library available for use in your program.

Two important libraries are the *machine* and *time* library.

- Machine contains routines specific to the microcontroller you're using, such as setting I/O pins to input, output setting the frequency and duty cycle for square waves, etc.
- Time contains wait routines.

Within *machine* is the function *Pin* - which controls whether a pin is input or output. Options are:

```
import machine

# Output
Button = machine.Pin(0, Pin.OUT)

# Inputs
LED0 = machine.Pin(6, Pin.IN)
LED1 = machine.Pin(7, Pin.IN, Pin.PULL_UP)
LED2 = maching.Pin(8, Pin.IN, Pin.PULL_DOWN)
```

The instruction *machine.Pin()* tells Python to use the routine *Pin* from library *machine.* This syntax allows different libraries to have identical function names without causing a conflict. It does get a little unwieldy, however. A shortcut assuming that there are no conflicts with the name *Pin()* is

```
1    from machine import Pin
2
3    Button = Pin(0, Pin.OUT)
4    LED0 = Pin(6, Pin.IN)
5    LED1 = Pin(7, Pin.IN, Pin.PULL_UP)
6    LED2 = Pin(8, Pin.IN, Pin.PULL_DOWN)
```

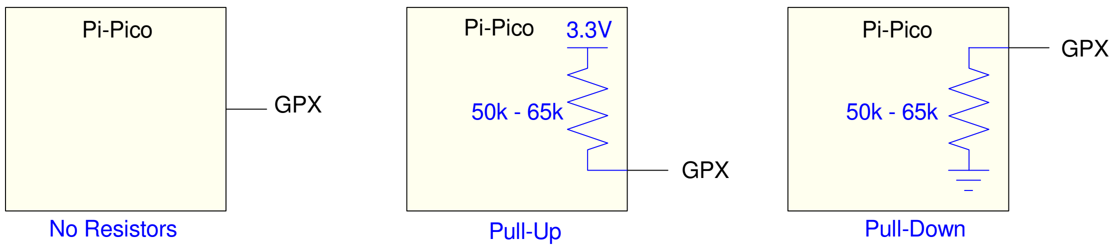
Line 3: This instruction tells Python that pin #0 is an output pin.

- Logic 0 outputs 0V
- Logic 1 outputs 3.3V

capable of sourcing or sinking up to 12mA.

Line 4 - 6: These instructions tell Python that pins 6, 7, and 8 are input pins.

- Pin 6 has a floating input: it is up to the hardware to assure the voltage is either 0V or 3.3V
- Pin 7 has an internal pull-up resistor. If left floating, pin 7 will read logic 1.
- Pin 8 has an internal pull-down resistor. If left floating, pin 8 will read logic 0.



Input pins can be floating, pulled high, or pulled-low.

The value of I/O pins can be read and written to as (# is a comment in Python)

```
# read
Y = Button.value()

# write
LED0.toggle()       # toggle LED0 on/off
LED0.value(1)       # set LED0
LED0.value(0)       # clear LED0
LED0.low()          # clear LED0
LED0.high()         # set LED0
```

Note: For outputs,

- 0 is off (false)
- anything else is on (true)

The *time* library contains wait routines:

- sleep(x): pause x seconds. x can be a floating-point number
- sleep_ms(x): pause x milliseconds. x must be an integer
- sleep_us(x): pause x microseconds. x must be an integer.

For example, you can read the value of the push buttons:

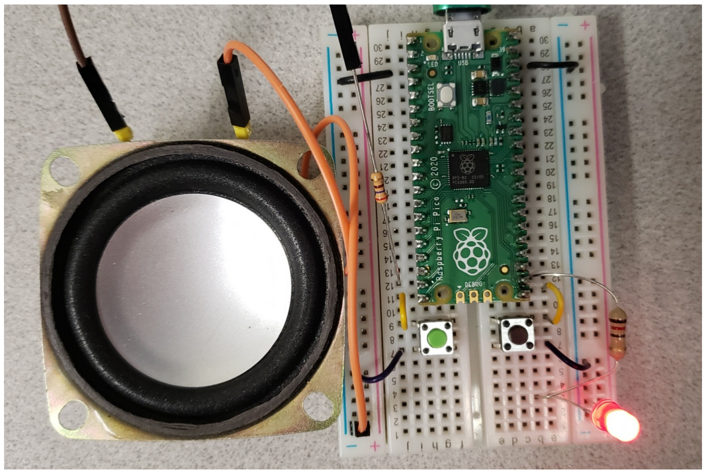
- When button B0 (green button) is pressed, X reads as 0
- When button B1 (brown button) is pressed, Y reads as 0
-

Open Save Run Stop

```
from machine import Pin
from time import sleep_ms

B0 = Pin(15, Pin.IN, Pin.PULL_UP)
B1 = Pin(16, Pin.IN, Pin.PULL_UP)

while(1):
    X = B0.value()
    Y = B1.value()
    print(X, Y)
    sleep_ms(500)
```

shell

```
MPY: soft reboot
1 1
1 1
0 1
0 1
1 1
1 1
1 0
1 0
1 1
1 1
1 1
```



Status of buttons are displayed in the shell window
(shows off better in the video)

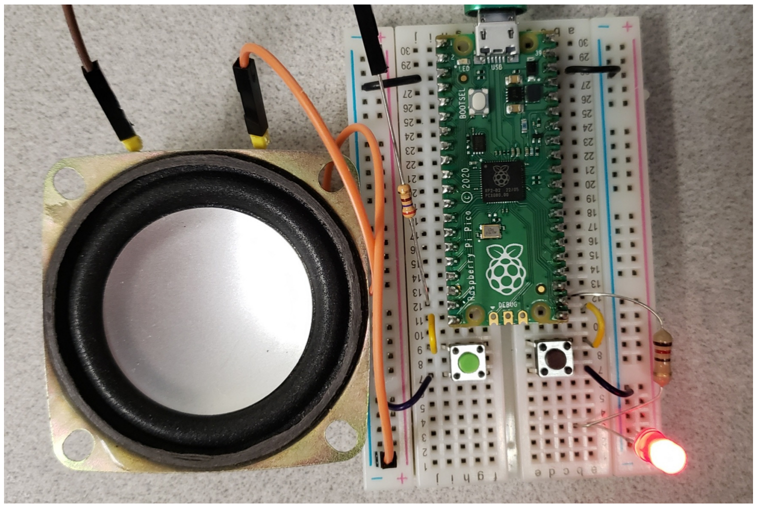
Turn on a light for two seconds: A program which turns on an LED for two seconds would be:

Open Save Run Stop

```
from machine import Pin
from time import sleep

LED = Pin(17, Pin.OUT)

LED.value(1)
print('LED On')
sleep(2)
LED.value(0)
print('LED Off')
```

Shell

```
>>>
    LED On
    LED Off
```



LED turn on for two seconds
(shows off better in the video)

if and while statements

While Python is similar to Matlab, there are some differences. One main difference is

- Python does not use *end* statements to terminate if-statements and loops
- Instead, Python uses indentation.

When you start a while loop, all of the subsequent code which is indented is within that while loop. The end of the while-loop is indicated by removal of the indentation.

Similar with if-statements: all subsequent code that is within that if-statement must be indented. The end of the if-statement is indicated by the removal of the indentation.

For example, the following code consists of two separate if statements:

```
if(a > 2):
    c = 3
if(b > 3):
    c = 4
```

If you want to next the if-statements, the second set needs to be indented

```
if(a > 2):
    c = 3
    if(b > 3):
        c = 4
```

If - else-if, else statements are also supported in Python

```
if(TempC > 40):
    print('very hot')
elif(TempC > 30):
    print('hot')
elif(TempC > 20):
    print('comfy')
else:
    print('cool')
```

For example, the following program uses two buttons to turn on and off a light:

- B0 (GP15) turns on the light
- B1 (GP16) turns off the light

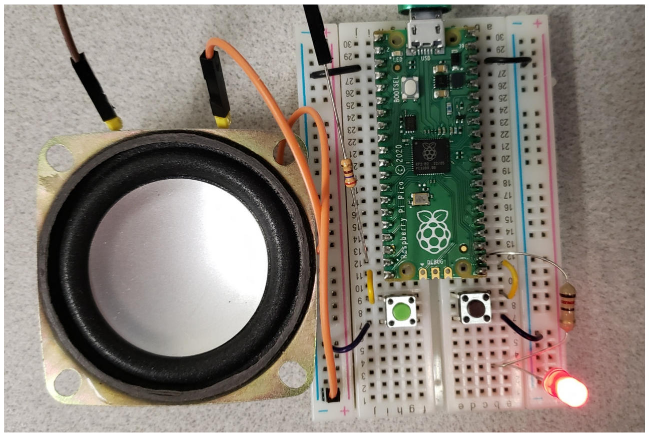
Open Save Run Stop

```
from machine import Pin
from time import sleep_ms

B0 = Pin(15, Pin.IN, Pin.PULL_UP)
B1 = Pin(16, Pin.IN, Pin.PULL_UP)
LED = Pin(17, Pin.OUT)
Spkr = Pin(14, Pin.OUT)

while(1):
    X = B0.value()
    Y = B1.value()
    if(X == 0):
        LED.value(1)
    if(Y == 0):
        LED.value(0)

    print(X, Y, LED.value())
    sleep_ms(100)
```

The shell window shows the button status (0 = pressed) and the LED status (1 = on)

shell

```
X   Y   LED
1   1   0       LED starts off as off
1   1   0
0   1   1       when X is pressed, the LED turns on
0   1   1
1   1   1
1   0   0       when Y is pressed, the LED turns off
1   0   0
1   1   0
```



LED turns on when you press GP15 (green)
LED turns off when you press GP16

Two-Key Piano:

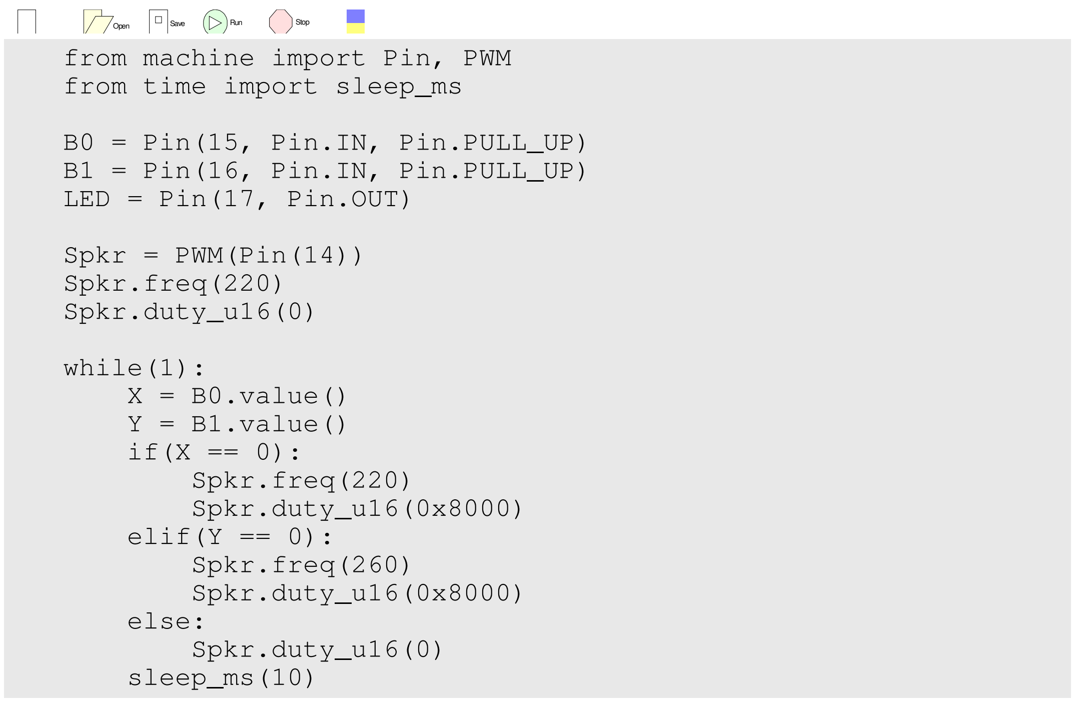
You can also use if-statements to turn your Pico into a two-key piano. The function *PWM* in the machine library outputs a pulse-width-modulated square-wave.

- Spkr = PWM(Pin(14)) sets I/O pin GP14 to an output pin which outputs a square wave
- Spkr.freq(220) sets the frequency of the square wave to 220Hz
- Spkr.duty_u16(x) sets the duty cycle of the square wave
  - x = 0: 0% duty cycle (off)
  - x = 0x8000: 50% duty cycle square wave (on)
  - x = 0xFFFF: 100% duty cycle square wave

The piano works as:

- Button B0 plays 220Hz when pressed
- Button B1 plays 260Hz when pressed
- When neither button is pressed, the speaker is off

Code:



```
from machine import Pin, PWM
from time import sleep_ms

B0 = Pin(15, Pin.IN, Pin.PULL_UP)
B1 = Pin(16, Pin.IN, Pin.PULL_UP)
LED = Pin(17, Pin.OUT)

Spkr = PWM(Pin(14))
Spkr.freq(220)
Spkr.duty_u16(0)

while(1):
    X = B0.value()
    Y = B1.value()
    if(X == 0):
        Spkr.freq(220)
        Spkr.duty_u16(0x8000)
    elif(Y == 0):
        Spkr.freq(260)
        Spkr.duty_u16(0x8000)
    else:
        Spkr.duty_u16(0)
    sleep_ms(10)
```

For-Loops

A for-loop starts with the first number and increments. Some variations are:

range(a, b):

- start at a,
- end when you reach b or higher (different than Matlab)

For example, range(0,5) counts from 0 to 4

```
for i in range(0,5):
    print(i, i*i)
```

Shell

```
>>>
    0     0
    1     1
    2     4
    3     9
    4    16
```

If you add a third term, this is the step-size

```
for i in range(0,5, 2):
    print(i, i*i)
```

Shell

```
>>>
    0     0
    2     4
    4    16
```

If you include an array, the for-loop steps through the array

```
for i in [1,3,5,7,11]:
    print(i, i*i)
```

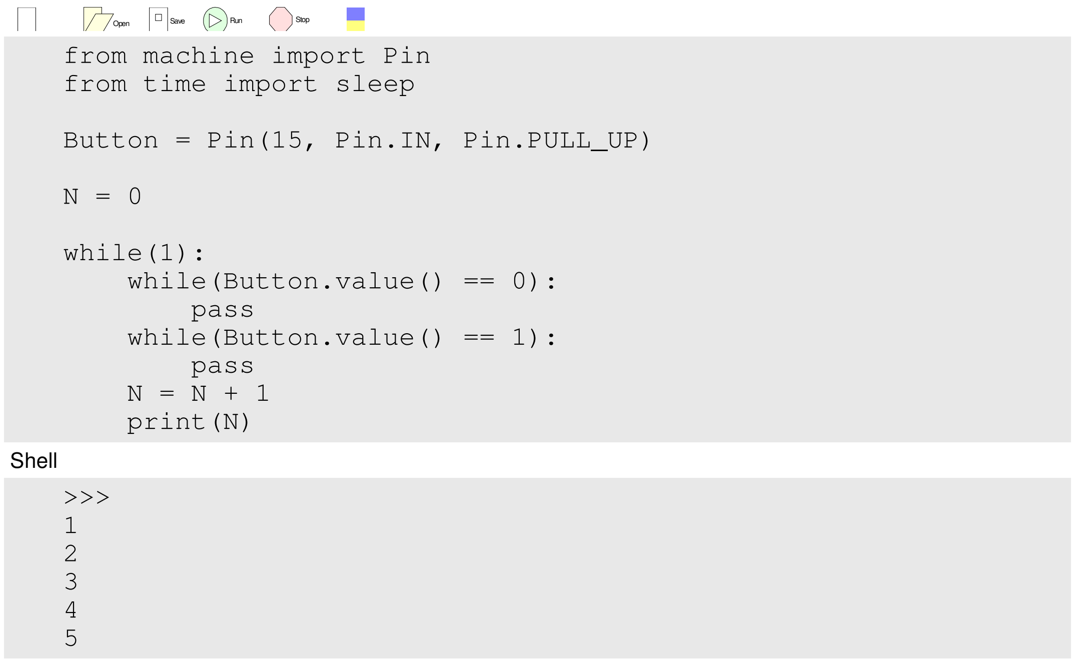
Shell

```
>>>
    1     1
    3     9
    5     25
    7     49
   11     121
```

Counter in Python (take 1)

As an example, write a Python program that counts how many times a button was pressed. Assume the hardware is:

In code:



```
from machine import Pin
from time import sleep

Button = Pin(15, Pin.IN, Pin.PULL_UP)

N = 0

while(1):
    while(Button.value() == 0):
        pass
    while(Button.value() == 1):
        pass
    N = N + 1
    print(N)
```

```
>>>
1
2
3
4
5
```

Subroutines in Python

In MicroPython, subroutines are defined by the keyword *def*, sort for *define*. Ths simplest example would be a routine which

- is passed nothing,
- returns nothing, and
- simply prints 'hello' when called:

When you press the *run* command

- Python installs the subroutine defined as *SayHello*
- It then runs the main routine (instruction following all of the definitions)

```
def SayHello():
    print('hello')

# Start of main routine
SayHello()
```

shell

```
>>>
hello
```

In this example, note that

- The subroutine is called *SayHello*
- Nothing is passes to this routine as indicated by the ()
- The definition is terminated with a colon (:)
- The code within the subroutine must be indented as per the Python standard

You can pass parameters to subroutines. For example, if you want to display numbers from 0..N, you could write a routine like the following:

```
def CountToN(N):
    for i in range(1,N+1):
        print(i)

# Start of main routine
CountToN(5)
```

Thonny Program Window

```
>>>
1
2
3
4
5
```

You can also pass multiple parameters by simply including them in the definition

```
def Multiply(A, B):
    C = A * B
    print(A, ' * ', B, ' = ',C)

# Start of main routine
Multiply(4,6)
```

shell

```
>>>
4 * 6 = 24

>>> Multiply(8,7)
8 * 7 = 56
```

Subroutines in Python can only return zero, one, or several variables. That variable could be an array, a matrix, or a class object however - so that's not very limiting.

Example where one number is returned:

```
    # Example of Returning One Number
    def Multiply(A, B):
        C = A * B
        return(C0)

    # Start of main routine
    X = Multiply(4,6)
    print(X)
```

shell

```
    >>>
    24

    >>> C = Multiply(8,7)
    >>> print(C)
    56
```

Example where four numbers are returned:

```
    # Example of Returning four Numbers
    def Operate(A, B):
        C0 = A + B
        C1 = A - B
        C2 = A * B
        C3 = A / B
        return([C0, C1, C2, C3])

    # Start of main routine
    X = Operate(4,6)
    print(X)
```

shell

```
    >>>
    [10, -2, 24, 0.666667]

    >>> C = Operate(8,7)
    >>> print(C)
    [15, 1, 56, 1.4142857]

    >>> [a, b, c, d] = Operate(8,7)
    >>>> print(a, b, c, d)
    15, 1,  56,  1.4142857
```

Going back to the counter program, you could clean up the code with a subroutine:

```
from machine import Pin
from time import sleep

Button = Pin(0, Pin.IN, Pin.PULL_UP)
g = Pin(8, Pin.OUT)
y = Pin(7, Pin.OUT)
r = Pin(6, Pin.OUT)

def Display(X):
    g.value(X & 0x01)
    y.value(X & 0x02)
    r.value(X & 0x04)

N = 0
Display(N)

while(1):
    while(Button.value() == 0):
        pass
    while(Button.value() == 1):
        pass
    N = (N + 1) % 8
    Display(N)
    print(N, r.value(), y.value(), b.value())
```

Shell

```
>>>
1    0    0    1
2    0    1    0
3    0    1    1
4    1    0    0
5    1    0    1
```

Program Execution on Startup

Make your Pi-Pico blink three times at 2Hz on power-up

- On for 100ms
- Off for 400ms
- repeat 3x

First, create a program (assume GP16 has an LED attached)

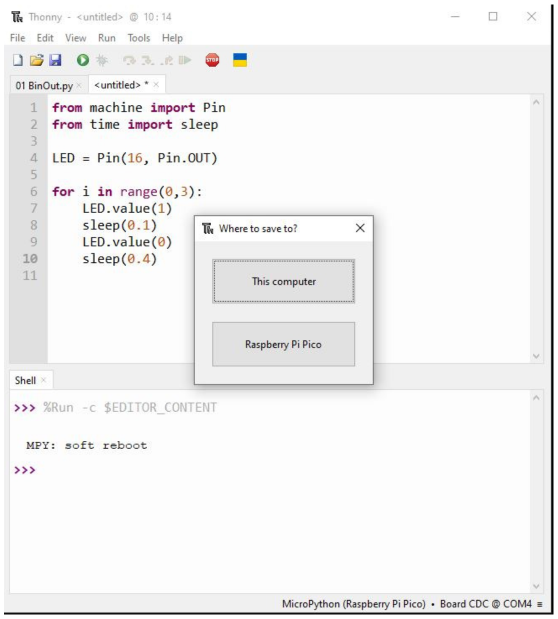
```
from machine import Pin
from time import sleep

LED = Pin(16, Pin.OUT)

for i in range(0,3):
   LED.value(1)
   sleep(0.1)
   LED.value(0)
   sleep(0.4)
```

Shell

Once this runs,

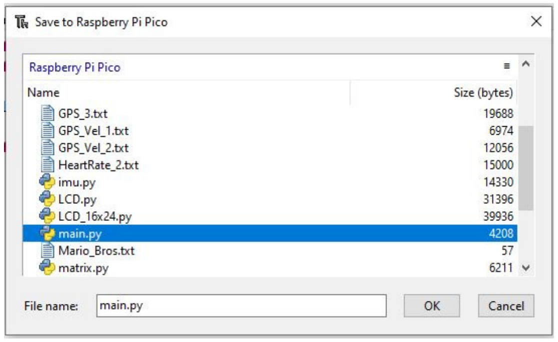
- Go to File Save As
- Select save to Raspberry Pi Pico
- Save as *main.py*

On power up, this program will execute.


Thonny - <untitled> @ 10 : 14
File Edit View Run Tools Help
01 BinOut.py
<untitled> *
Where to save to?
This computer
Raspberry Pi Pico
Shell
MicroPython (Raspberry Pi Pico) • Board CDC @ COM4

```
from machine import Pin
from time import sleep

LED = Pin(16, Pin.OUT)

for i in range(0,3):
    LED.value(1)
    sleep(0.1)
    LED.value(0)
    sleep(0.4)
```

```
>>> %Run -c $EDITOR_CONTENT

  MPY: soft reboot
>>>
```



Name	Size (bytes)
GPS_3.txt	19688
GPS_Vel_1.txt	6974
GPS_Vel_2.txt	12056
HeartRate_2.txt	15000
imu.py	14330
LCD.py	31396
LCD_16x24.py	39936
main.py	4208
Mario_Bros.txt	57
matrix.py	6211

File name: main.py

Appendix: MicroPython Syntax

Assigning values to variables:

```
X = 123           decimal 123
X = 0x123         hex 123
x, y, z = 1, 2, 3
X = [1,2,3,4,5]   matrix or array
X = range(1,6)    same matrix
X = [[1,2],[3,4]] 2x2 matrix
X = [0]*100       1x100 matrix of zeros (different syntax than Matlab)
```

Operations

```
+                 add
-                 subtract
*                 multiply
/                 divide (result is usually a float)
//                divide and round down (result is integer)
%                 modulus (remainder)
**                raise to the power

X.append(6)   append 6 to the end of array X
```

Logic Operations

```
&                 logical AND (bitwise)
|                 logical OR (bitwise)
^                 logical XOR (bitwise)
>>                shift right
<<                shift left
```

# comment statement

```
# this is a comment statement
```

Conditionals:

```
X > Y
X < Y
X >= Y
X == Y
X != Y
```

Converting variable types:

```
int(X)    convert to an integer, round down
round(X)  round to nearest integer
float(X)  convert to a floating point number
```

note: Python automatically adjusts variable types - you don't need to declare them like you do in C. For example:

```
>>> X = 3         X is automatically treated like an integer
>>> Y = 4         Y is automatically treated like an integer
>>> Z = X/Y       Z becomes a float  (0.75)
>>> Z = X//Y      Z is an integer (0)
```

print() Information can be sent to the shell window using a *print()* statement

```
>>> print('Hello World')
Hello World

>>> X = 2**0.5
>>> print('X = ',X)
X =  1.414214
```

X = input() Information can be passed to your program using the *input()* statement. For example, prompt the user to input a number for X:

```
>>> X = input('Type in a number')
```

This will result in X being a string (typing in *Hello World* is valid). If you want to receive the input as a number, convert the result as:

```
>>> X = int( input('Type in a number') )
>>> X = float( input('Type in a number') )
```

When writing to the shell, numbers can be formatted if desired. Examples follow:

```
>>> msg = '27 in binary = {:b}'.format(27)
>>> msg
'27 in binary = 11011'

>>> msg = '27 in binary = {:b}'.format(27)
>>> msg
'27 in binary = 11011'
>>> msg = '27 in hex = {:X}'.format(27)
>>> msg
'27 in hex = 1B'

>>> msg = '0x2134 in decimal = {:d}'.format(0x1234)
>>> msg
'0x2134 in decimal = 4660'

>>> msg = '123.4567 rounded to 2 decimal = {:.2f}'.format(123.4567)
>>> msg
'123.4567 rounded to 2 decimal = 123.46'

>>> msg = '123.4567 rounded to 2 decimal = {:.2e}'.format(123.4567)
>>> msg
'123.4567 rounded to 2 decimal = 1.23e+02'

>>> msg = '79/255 = {:.2%}'.format(79/255)
>>> msg
'79/255 = 30.98%'
```