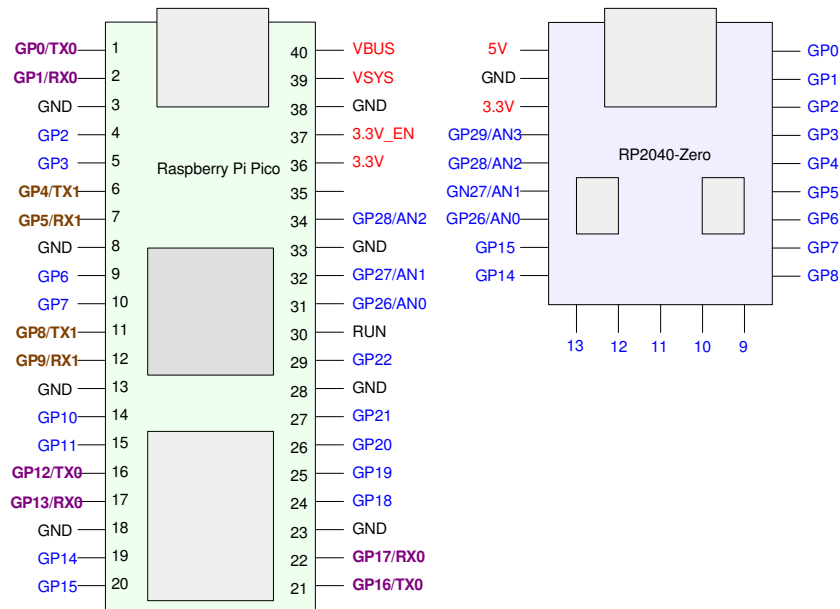


# Designs using a Raspberry Pi-Pico & Python



## Introduction

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:**

**Power:** Power can be provided to a Pi-Pico several ways:

- USB: The USB cable provides 5V to the Pi-Pico
- VSYS (pin 39): Provide 1.8V to 5.5V to VSYS.

Regardless of how you power the Pi-Pico, two output voltages are available for the rest of your circuit:

- VBUS (pin 40): Outputs 5V
- 3.3V (pin 36) Outputs 3.3V

**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

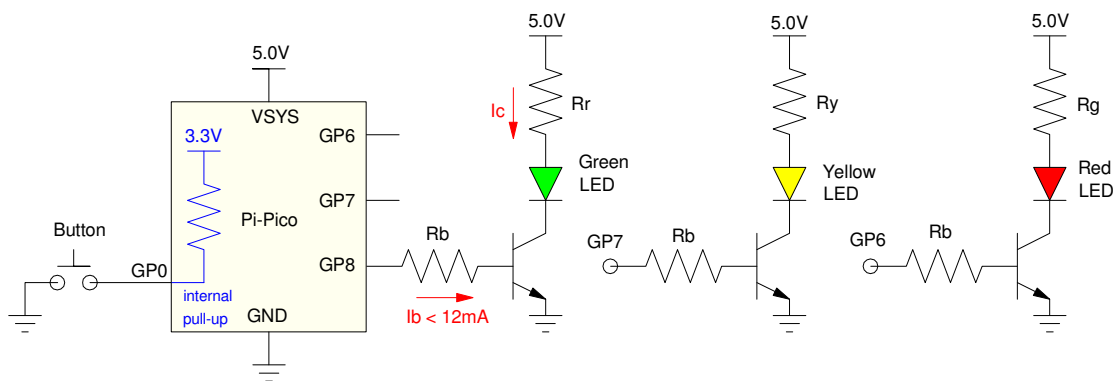
**Do not connect 5.0V to the Pi-Pico's input pins. This may damage the Pi-Pico.**

Each input pin can also be connected to an internal pull-up or pull-down resistor.

For example, suppose you want to connect your Pi-Pico to some inputs and outputs:

- Power: from the USB cable
- Inputs: Push button connected to GP0
- Outputs: Three LEDs connected to GP6, GP7, and GP8

The hardware could be:



Hardware for connecting a Pi-Pico to a push button (input) and there LEDs (output)  
 VSYS does not need to be connected to +5V if powered from the USB cable

## Software: Thonny and MicroPython

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

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

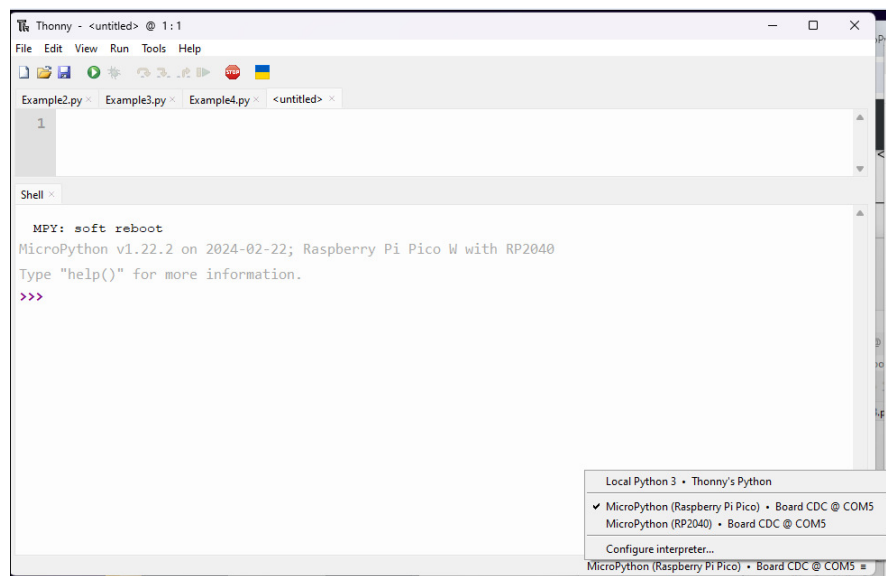
among others. In this lecture, we focus on MicroPython.

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



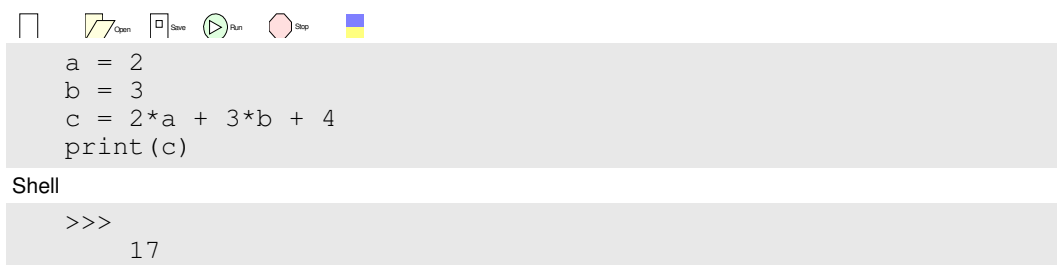
Python is similar to Matlab: you can type commands directly in the shell window. For example, you can use Python like a calculator:



The screenshot shows a Python IDE interface. At the top, there are icons for Open, Save, Run, and Stop. Below these icons is a grey bar labeled "Shell". Inside the shell, the following code is entered and executed:

```
>>> 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)



The screenshot shows a Python IDE interface. At the top, there are icons for Open, Save, Run, and Stop. Below these icons is a grey bar containing the following code:

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

Below the script window is another grey bar labeled "Shell". Inside the shell, the following code is entered and executed:

```
>>>
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 ratio 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 = machine.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

```
from machine import Pin

Button = Pin(0, Pin.OUT)
LED0 = Pin(6, Pin.IN)
LED1 = Pin(7, Pin.IN, Pin.PULL_UP)
LED2 = Pin(8, Pin.IN, Pin.PULL_DOWN)
```

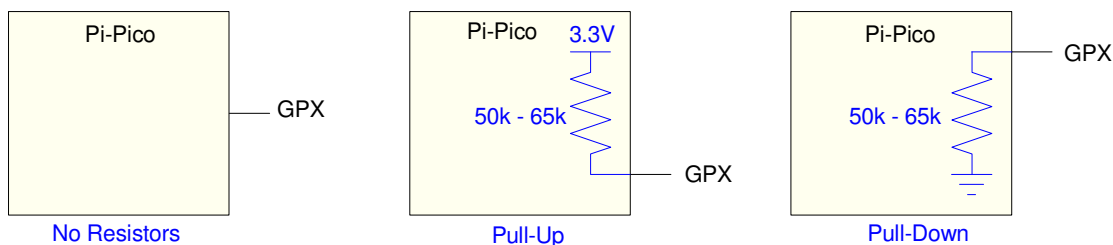
**Line 3:** This instruction tells Python that pin #0 is an output pin.

- 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
```

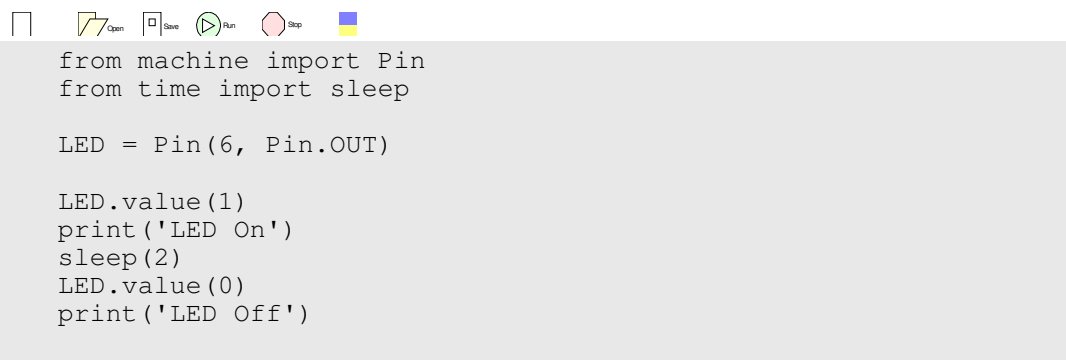
Note: For outputs,

- 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, a program which turns on LED0 for two seconds would be:



```
from machine import Pin
from time import sleep

LED = Pin(6, Pin.OUT)

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

Shell

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

## Loops

Python also supports for-loops, while-loops, and if-statements like Matlab. The syntax in Python is slightly different, however.

- In Matlab, carriage returns and indentation is decorative. They help the programmer write understandable code but have no impact on the program's execution.
- In Python, carriage returns and indentation have meaning.
- In Matlab, for-loops, while-loops, and if-statements are terminated with *end* statements
- In Python, there are no *end* statements. Indentation tells you where the loop ends.

The format for these statements in Python are as follows:

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

x = 3
while(x > 0):
    x -= 1

a = b = 4
if(a > b):
    print('a is greater than b')
elif(a == b):
    print('a is equal to b')
else:
    print('a is less than b')
```

Note that

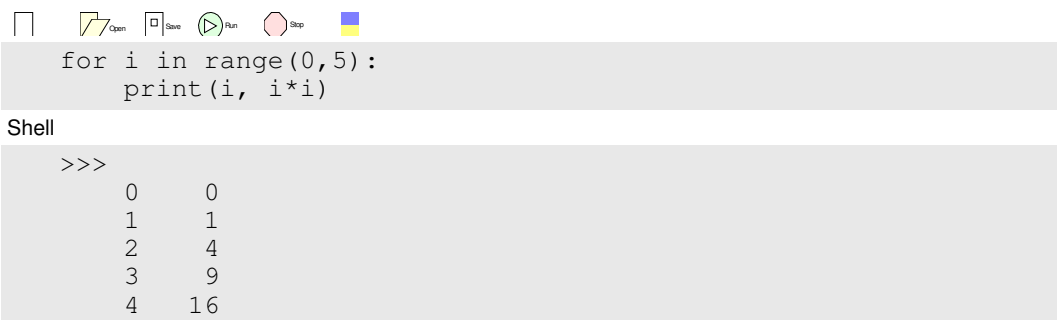
- Each loop starts with a statement that ends in a colon. This tells Python a loop is starting.
- The following lines are indented. (four spaces is standard but Python doesn't care)
- The end of the loop is indicated by removing the indentation

**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



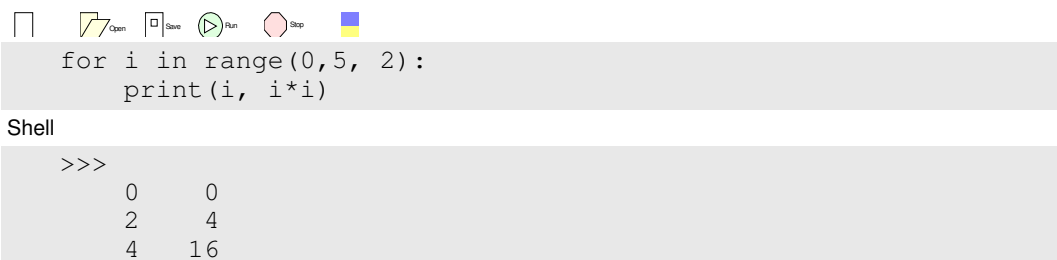
The screenshot shows a Python IDE interface with a toolbar at the top containing icons for file operations (New, Open, Save, Run, Stop) and a color palette. Below the toolbar, a code editor contains the following Python code:

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

Below the code editor, a shell window displays the output of the code:

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

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



The screenshot shows a Python IDE interface with a toolbar at the top containing icons for file operations (New, Open, Save, Run, Stop) and a color palette. Below the toolbar, a code editor contains the following Python code:

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

Below the code editor, a shell window displays the output of the code:

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

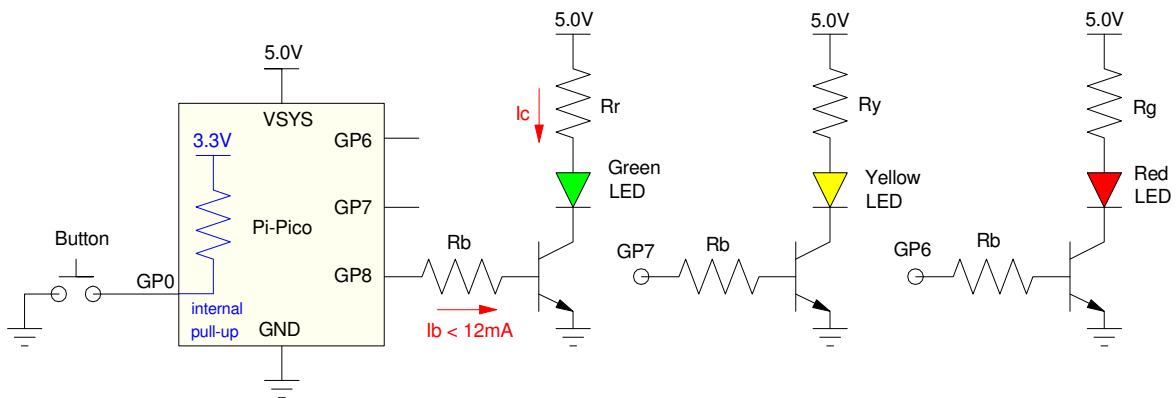
If you include an array, the for-loop steps through the array

```

    [ ] [ ] [ ] [ ] [ ] [ ]
    for i in [1,3,5,7,11]:
        print(i, i*i)
    Shell
    >>>
        1    1
        3    9
        5   25
        7   49
        11  121
    
```

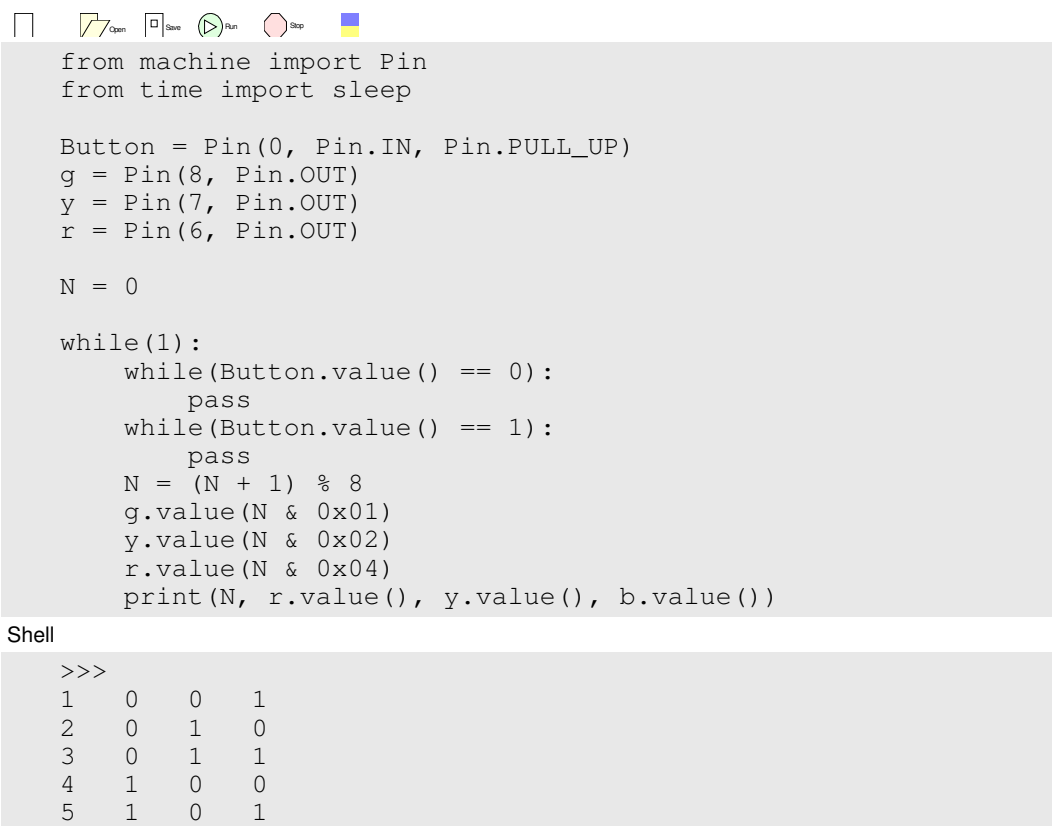
### Example: Counter in Python (take 1)

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

N = 0

while(1):
    while(Button.value() == 0):
        pass
    while(Button.value() == 1):
        pass
    N = (N + 1) % 8
    g.value(N & 0x01)
    y.value(N & 0x02)
    r.value(N & 0x04)
    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

```

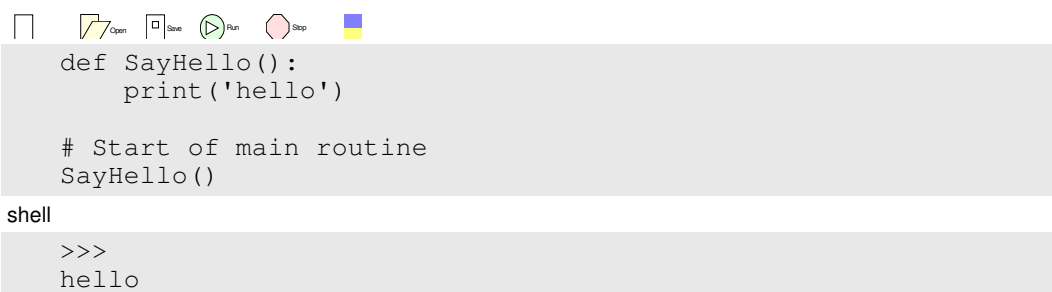
## Subroutines in Python

In MicroPython, subroutines are defined by the keyword *def*, sort for *define*. The 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 defed as *SayHello*
- It then runs the main routine (instruction following all of the definitions)



```

def SayHello():
    print('hello')

# Start of main routine
SayHello()

```

shell

```

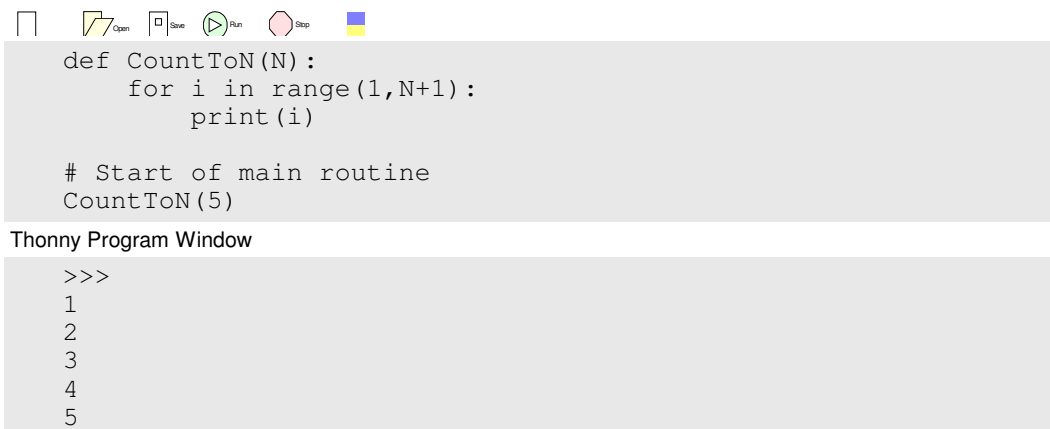
>>>
hello

```

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:



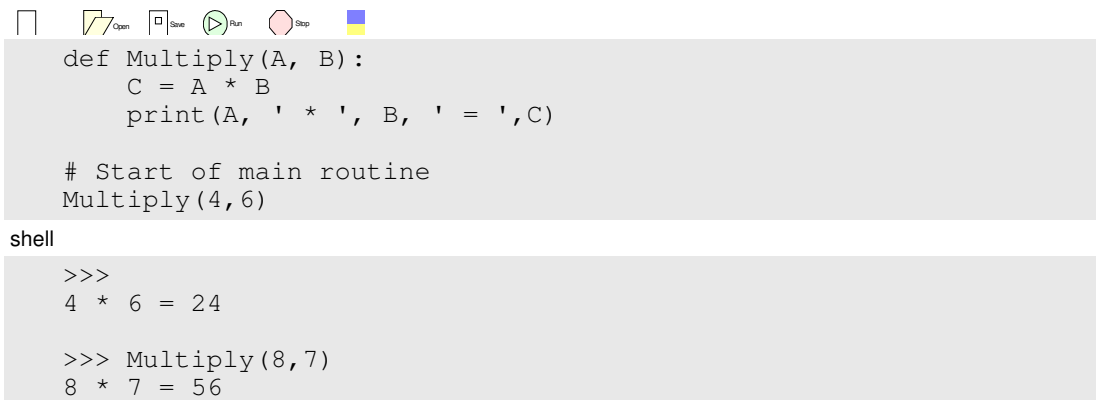
The screenshot shows the Thonny IDE interface. At the top, there are icons for File, Edit, Run, and Stop. The main editor window contains the following Python code:

```
def CountToN(N):  
    for i in range(1,N+1):  
        print(i)  
  
# Start of main routine  
CountToN(5)
```

Below the editor, the 'Thonny Program Window' shows the output of the program:

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

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



The screenshot shows the Thonny IDE interface. At the top, there are icons for File, Edit, Run, and Stop. The main editor window contains the following Python code:

```
def Multiply(A, B):  
    C = A * B  
    print(A, ' * ', B, ' = ', C)  
  
# Start of main routine  
Multiply(4, 6)
```

Below the editor, the 'shell' window shows the output of the program:

```
>>>  
4 * 6 = 24  
  
>>> Multiply(8, 7)  
8 * 7 = 56
```

Subroutines in Python can only return zero or one variable. 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(C)

# 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 in a matrix:



```
# 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]
```

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

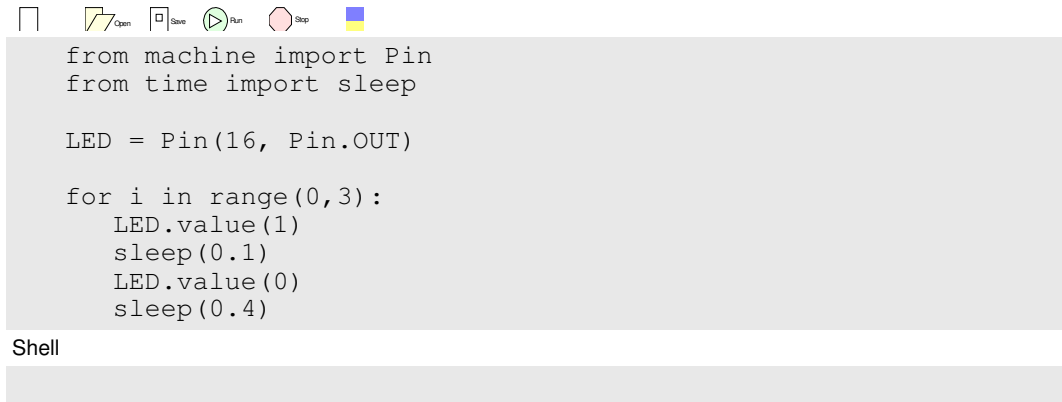
```
>>>
1    0    0    1
2    0    1    0
3    0    1    1
4    1    0    0
5    1    0    1
```

## 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)

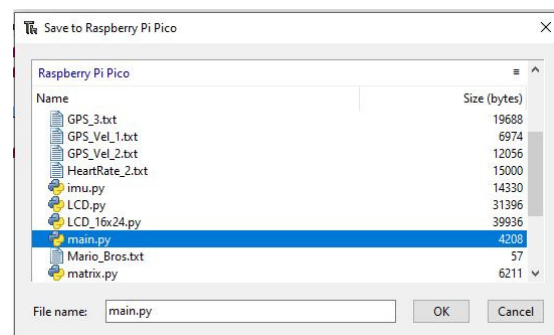
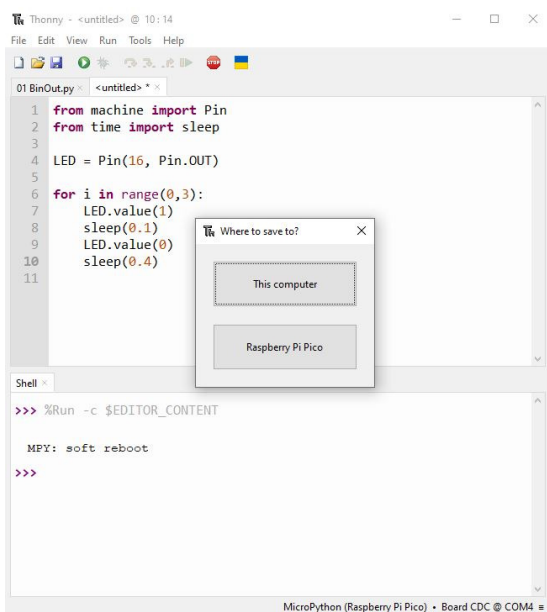
```

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.



## 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
```

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%'
```