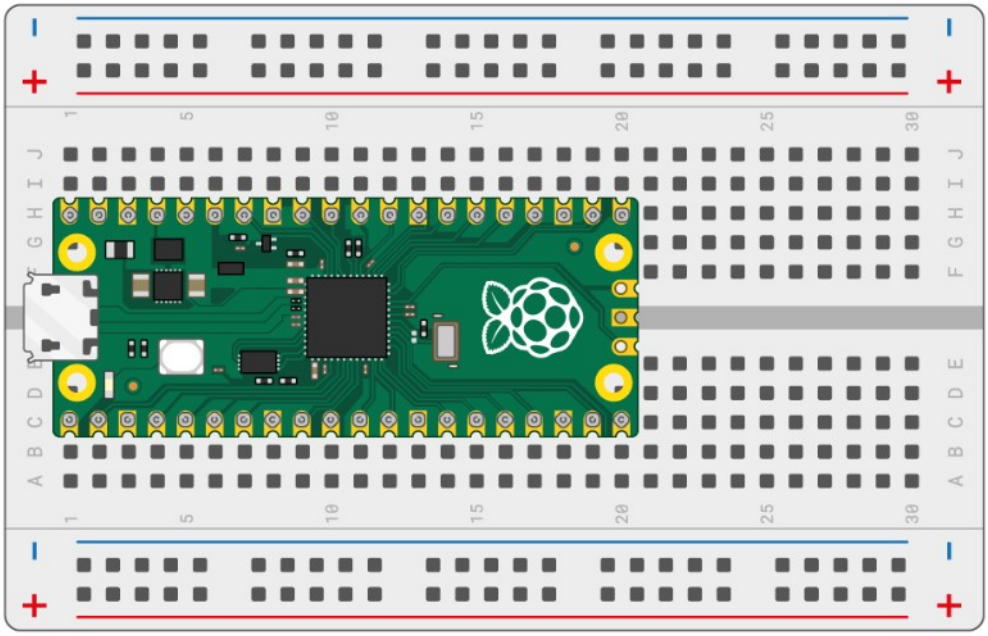
Part 12
-
MicroPython using Thonny

Version: 2021-01-25

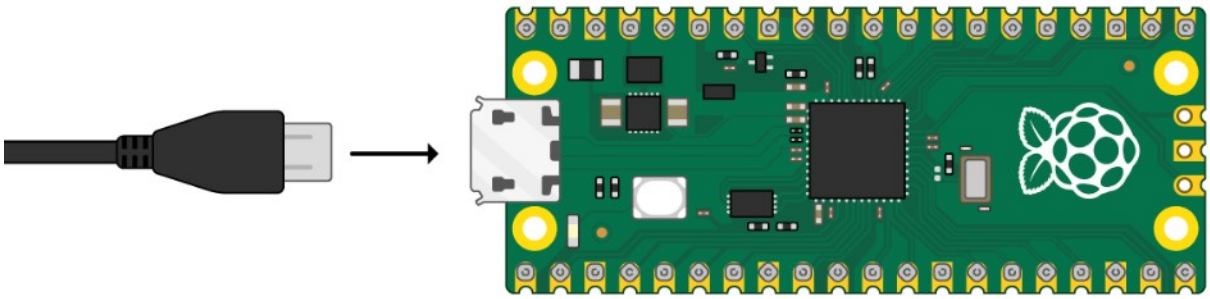
Introduction

Meet Pico

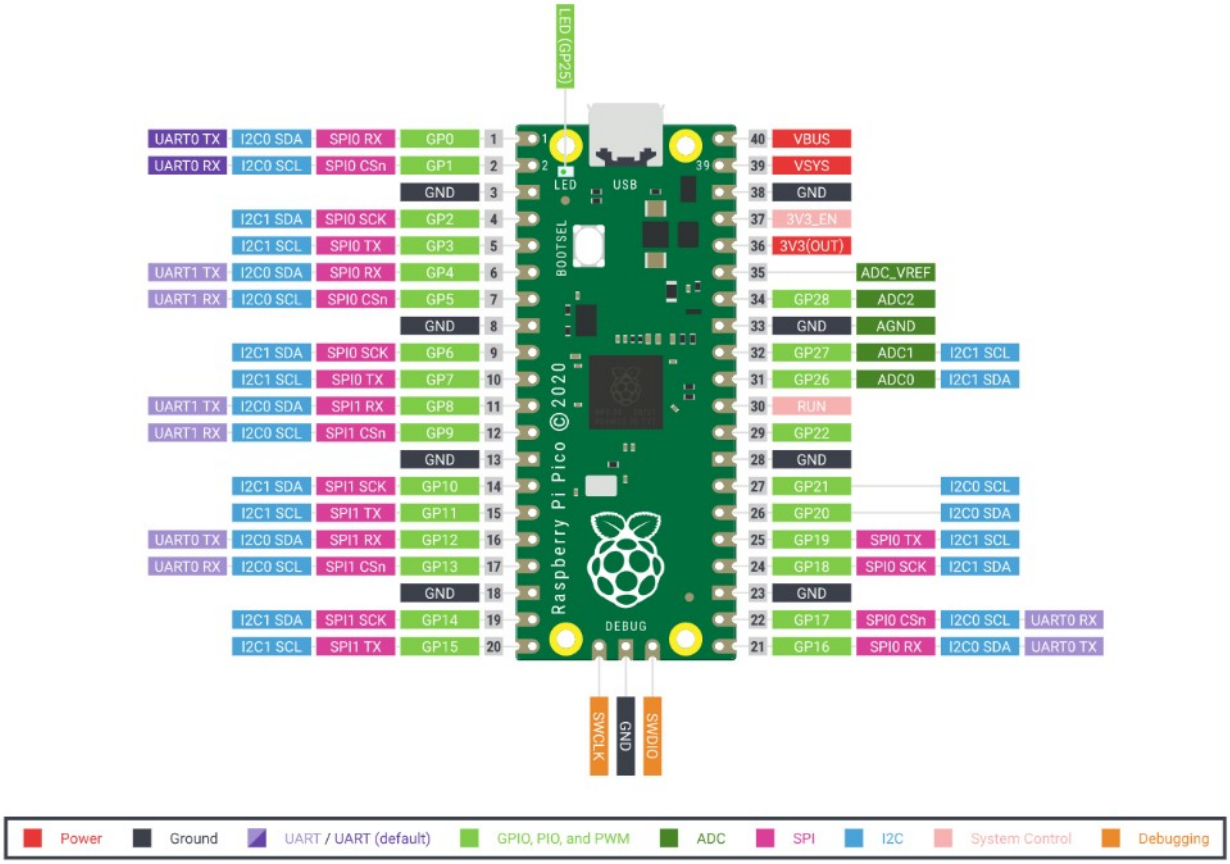
Hopefully your device has already had the header pins soldered on, but if not, do so now.
Put your Pico on the breadboard. Place it so that the two headers are separated by the ravine in the middle.



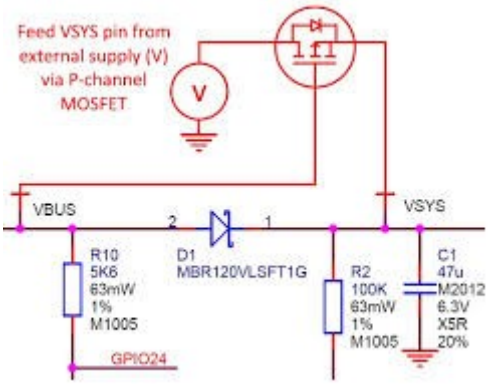
Plug your micro USB cable into the port on the left-hand side of the board and hook up the other side of the USB cable to a computer.
I will be using a Windows 10 computer to do the testing.

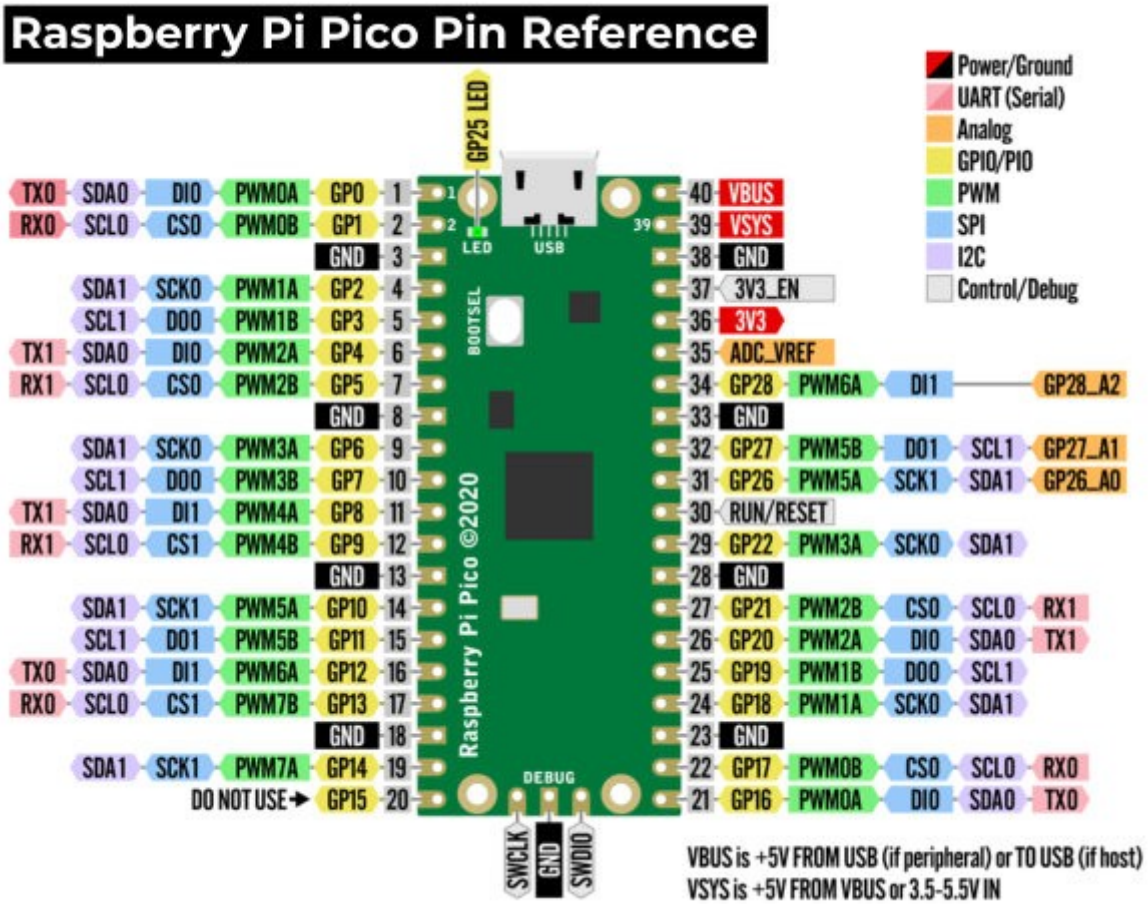


If you need to know the pin numbers for a Pico, you can refer to the following diagram.



VBUS is +5V from USB (if peripheral) or to USB (if host)
VSYS is +5V from VBUS or +3.5-5V IN



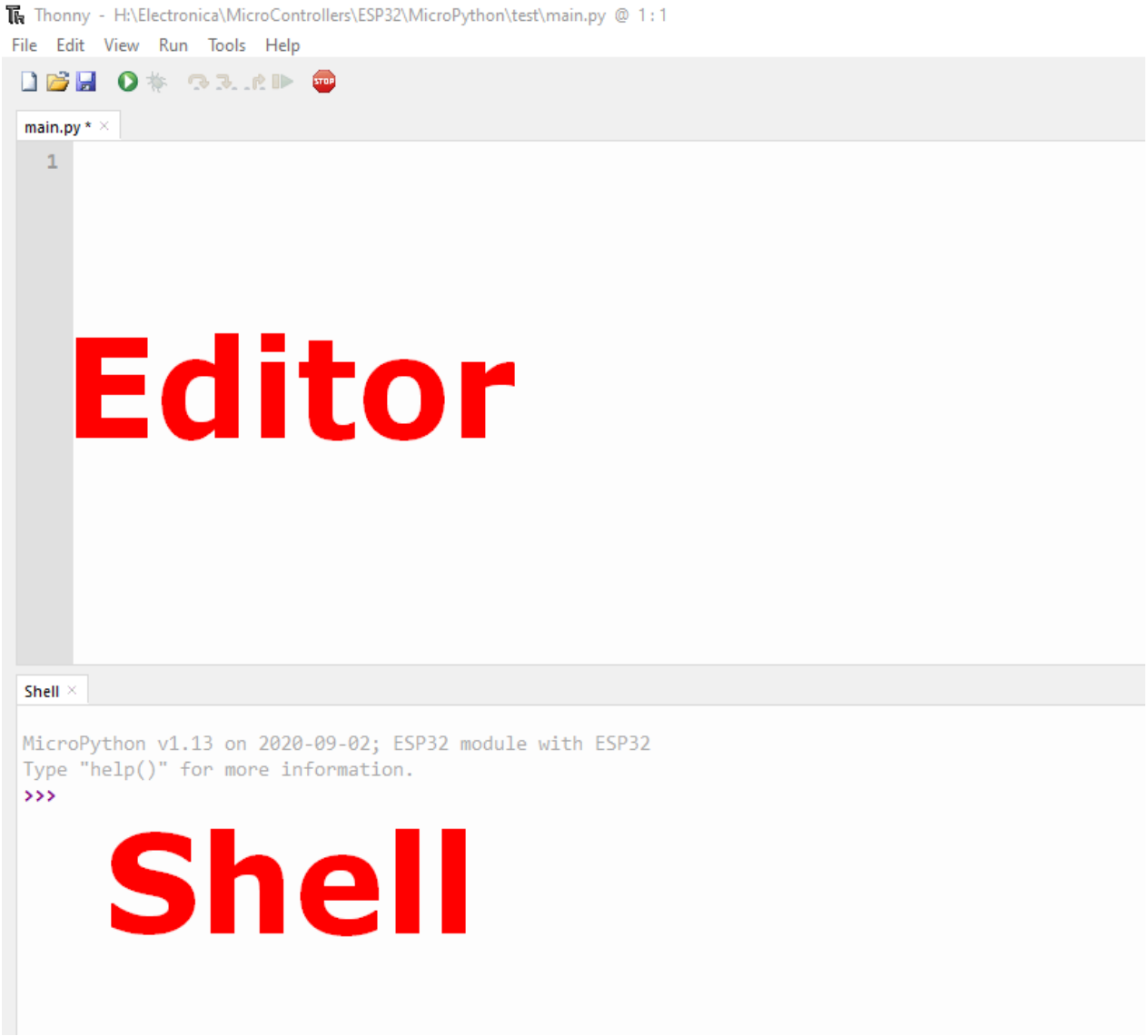


Make sure that MicroPython firmware is installed on the Pico (which includes installation of Thonny as well). If not, see previous document on MicroPython.

But first let’s take a look at Thonny IDE itself

Thonny IDE Overview

There are two different sections: the **Editor** and the **MicroPython Shell/Terminal**:



The **Editor** section is where you write your code and edit your *.py* files. You can open more than one file, and the Editor will open a new tab for each file.
On the **Shell** you can type commands to be executed immediately by your ESP board without the need to upload new files. The terminal also provides information about the state of an executing program, shows errors related with upload, syntax errors, prints messages, etc…

The Icons
Across the top you'll see several icons. You'll see an image of the icons below, with a letter above each one. We will use these letters to talk about each of the icons:

A: The paper icon allows you to create a new file. Typically in Python you want to separate your programs into separate files.

B: The open folder icon allows you to open a file that already exists on your computer. This might be useful if you come back to a program that you worked on previously.

C: The floppy disk icon allows you to save your code. Press this early and often. You'll use this later to save your first Thonny Python program.

D: The play icon allows you to run your code.

E: The bug icon allows you to debug your code. It's inevitable that you will encounter bugs when you're writing code. Bugs can come in many forms, sometimes appearing when you use inappropriate syntax and sometimes when your logic is incorrect. Thonny's bug button is typically used to spot and investigate bugs.

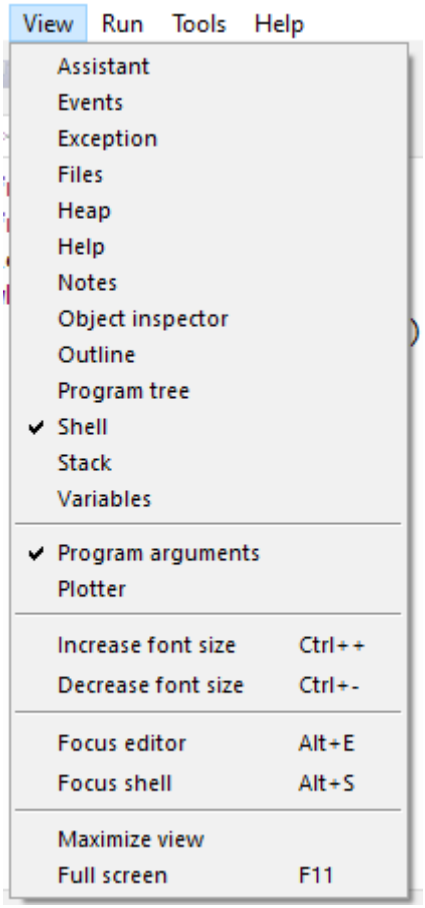
F-H: The arrow icons allow you to run your programs step by step. This can be very useful when you're debugging or, in other words, trying to find those nasty bugs in your code. These icons are used after you press the bug icon. You'll notice as you hit each arrow, a yellow highlighted bar will indicate which line or section Python is currently evaluating:

- The **F** arrow tells Python to take a big step, meaning jumping to the next line or block of code.
- The **G** arrow tells Python to take a small step, meaning diving deep into each component of an expression.
- The **H** arrow tells Python to exit out of the debugger.

I: The resume icon allows you to return to play mode from debug mode. This is useful in the instance when you no longer want to go step by step through the code, and instead want your program to finish running.

J: The stop icon allows you to stop running your code. This can be particularly useful if, let's say, your code runs a program that opens a new window, and you want to stop that program. You'll use the stop icon later in the tutorial.

Other UI Features
To see more of the other features that Thonny has to offer, navigate to the menu bar and select the *View* dropdown. You should see that *Shell* has a check mark next to it, which is why you see the Shell section in Thonny's application window:



Let's explore some of the other offerings, specifically those that will be useful

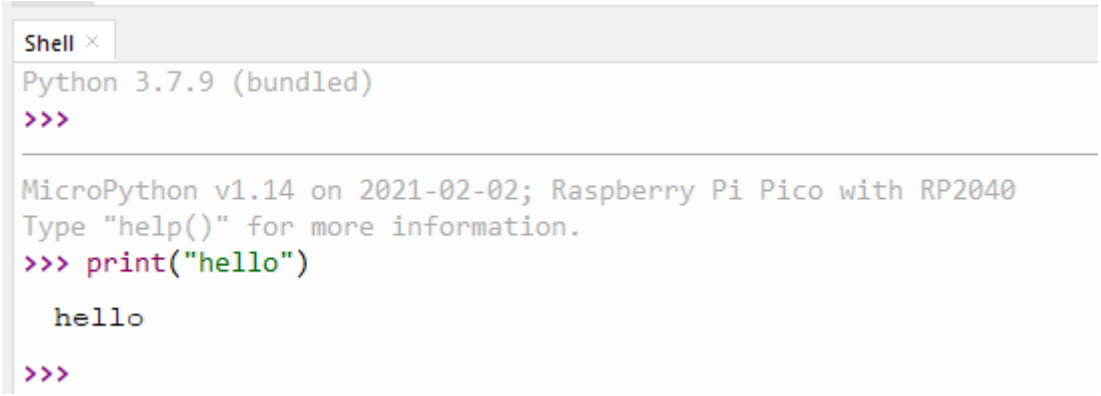
- **Help:** You'll select the *Help* view if you want more information about working with Thonny.
- **Variables:** This feature can be very valuable. A variable in Python is a value that you define in code. Variables can be numbers, strings, or other complex data structures. This section allows you to see the values assigned to all of the variables in your program.
- **Assistant:** The Assistant is there to give you helpful hints when you hit Exceptions or other types of errors.

The other features will become useful as you advance your skills. Check them out once you get more comfortable with Thonny!

Use the Shell
You will use the Thonny Shell to run some simple Python code on your Pico.
Make sure that your Pico is connected to your computer and you have selected the MicroPython (Pico) interpreter. Look at the Shell panel at the bottom of the Thonny editor.
Thonny is now able to communicate with your Pico using the REPL (read–eval–print loop), which allows you to type Python code into the Shell and see the output.
Now you can type commands directly into the Shell and they will run on your Pico.
Type the following command.

```
print("Hello")
```

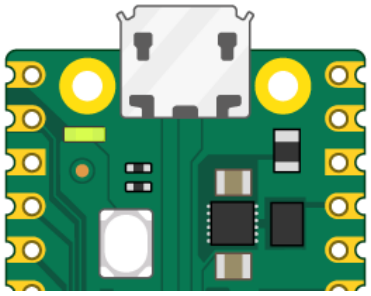
Tap the Enter key and you will see the output:



MicroPython adds hardware-specific modules, such as machine, that you can use to program your Pico. Let's create a machine.Pin object to correspond with the onboard LED, which can be accessed using GPIO pin 25.
If you set the value of the LED to 1, it turns on.
Enter the following code, make sure you tap Enter after each line.

```
from machine import Pin
led = Pin(25, Pin.OUT)
led.value(1)
```

You should see the onboard LED light up.



Type the code to set the value to 0 to turn the LED off.

```
led.value(0)
```

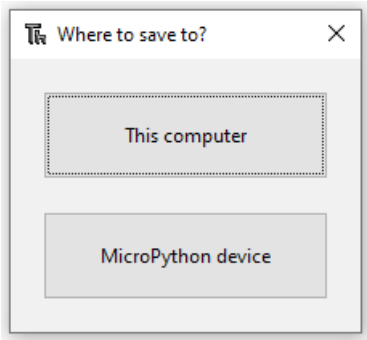
Turn the LED on and off as many times as you like.

Tip: You can use the up arrow on the keyboard to quickly access previous lines.
If you want to write a longer program, then it's best to save it in a file. You'll do this in the next step.

Blink the onboard LED
The Shell is useful to make sure everything is working and try out quick commands. However, it's better to put longer programs in a file. Thonny can save and run MicroPython programs directly on your Pico.
Click in the main editor pane of Thonny. Enter the following code to toggle the LED.

```
from machine import Pin
led = Pin(25, Pin.OUT)

led.toggle()
```

Click the Run button to run your code.
Thonny will ask whether you want to save the file on `This computer` or the `MicroPython device`. Choose `MicroPython device`.



Enter `blink.py` as the file name.

Tip: You need to enter the .py file extension so that Thonny recognises the file as a Python file.

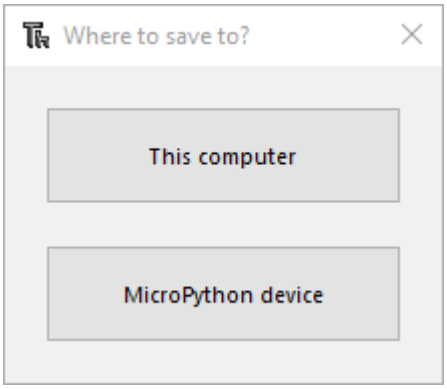
Thonny can save your program to your Pico and run it.
You should see the onboard LED switch between on and off each time you click the Run button.
You can use the Timer module to set a timer that runs a function at regular intervals.
Update your code so it looks like this:

```
from machine import Pin, Timer
led = Pin(25, Pin.OUT)
timer = Timer()

def blink(timer):
  led.toggle()

timer.init(freq=2.5, mode=Timer.PERIODIC, callback=blink)
```

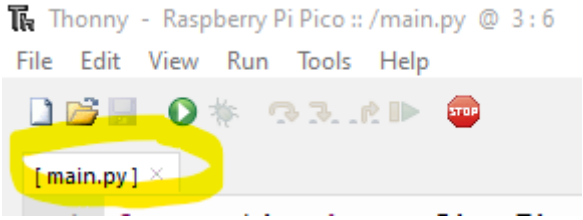
Click Run and your program will blink the LED on and off until you click the Stop button.

Creating the main.py file on your board

When you open Thonny IDE for the first time, the Editor shows an untitled file. Save that file as `main.py`. Simply, click the save icon. You will be asked to save the file on '`This computer`' or on '`MicroPython Device`'. Go for '`MicroPython Device`' and save it with the name `main.py`.



The Editor should now have a tab called `main.py`.
Notice the square brackets around main.py. This indicates the file is on the Pico. If square brackets are not present, file is on your computer.

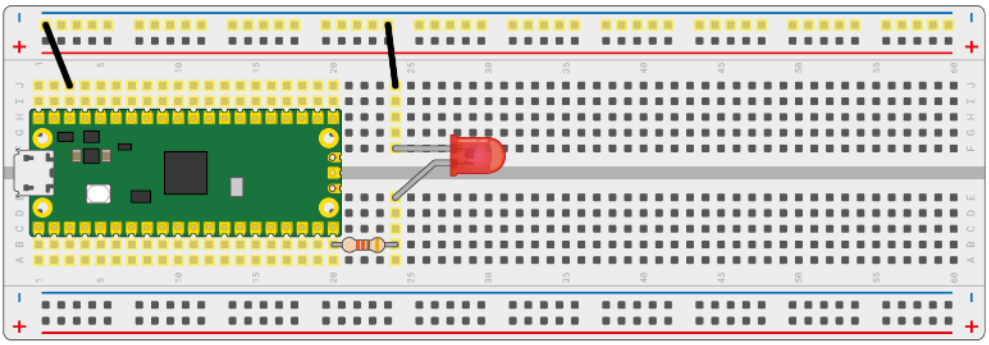


Note: uploading the code as `main.py` script will save the current file with the name `main.py` on the Pico, even if you have saved it in your computer with a different name. The same happens for the `boot.py` file.

Important: when the Pico restarts, first it runs the `boot.py` file and afterwards the `main.py`.

Use digital inputs and outputs

Now you know the basics, you can learn to control an external LED with your Pico, and get it to read input from a button. Use a resistor between about 50 and 330 ohms, an LED, and a pair of M-M jumper leads to connect up your Pico as shown in the image below.

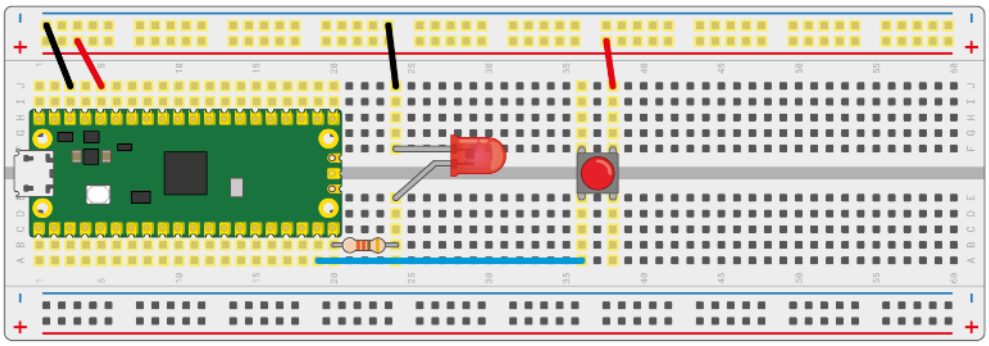


In this example, the LED is connected to pin 15. If you use a different pin, remember to look up the number in the pinout diagram in the Meet Pico section.
Use the same code as you did to blink the onboard LED, but change the pin number to 15.

```
from machine import Pin, Timer
led = Pin(15, Pin.OUT)
timer = Timer()

def blink(timer):
  led.toggle()

timer.init(freq=2.5, mode=Timer.PERIODIC, callback=blink)
```

Run your program and your LED should start to blink. If it's not working, check your wiring to be sure that the LED is connected.

Next, let's try and control the LED using a button. Add a button to your circuit as shown in the diagram below.



The button is on pin 14, and is connected to the 3.3V pin on your Pico. This means when you set up the pin, you need to tell MicroPython that it is an input pin and needs to be *pulled down*. Create a new file and add this code.

```
from machine import Pin
import time
led = Pin(15, Pin.OUT)
button = Pin(14, Pin.IN, Pin.PULL_DOWN)
while True:
  if button.value():
     led.toggle()
  time.sleep(0.5)
```

Run your code and then when you press the button, the LED should toggle on or off. If you hold the button down, it will flash.

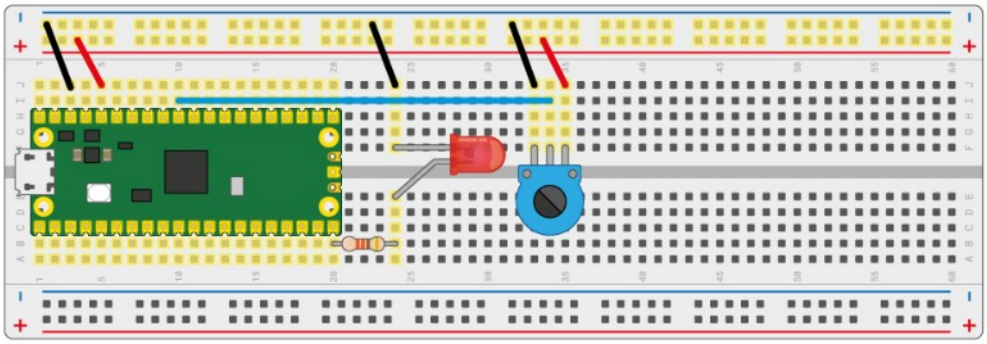
Control LED brightness with PWM

Pulse width modulation, allows you to give analogue behaviours to digital devices, such as LEDs. This means that rather than an LED being simply on or off, you can control its brightness. For this activity, you can use the circuit from the last step.
Open a new file in Thonny and add the following code.

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

pwm = PWM(Pin(15))

pwm.freq(1000)

while True:
    for duty in range(65025):
      pwm.duty_u16(duty)
      sleep(0.0001)
      for duty in range(65025, 0, -1):
        pwm.duty_u16(duty)
        sleep(0.0001)
```

Save and run the file. You should see the LED pulse bright and dim, in a continuous cycle. The frequency (`pwm.freq`) tells Pico how often to switch the power between on and off for the LED. The duty cycle tells the LED for how long it should be on each time. For Pico in MicroPython, this can range from 0 to 65025. 65025 would be 100% of the time, so the LED would stay bright. A value of around 32512 would indicate that it should be on for half the time. Have a play with the `pwm.freq()` values and the `pwm.duty_u16` values, as well as the length of time for the sleep, to get a feel for how you can adjust the brightness and pace of the pulsing LED.

Control an LED with an analogue input
Your Pico has input pins that can receive analogue signals. This means that instead of only reading the values of 1 and 0 (on and off), it can read values in between.
A potentiometer is the perfect analogue device for this activity.
Replace the button in your circuit with a potentiometer. Follow the wiring diagram below to connect it to an analogue pin.



In a new file in Thonny, you can first read the resistance of the potentiometer.
Add this code to a new file, and then run it.

```
from machine import ADC, Pin
import time

adc = ADC(Pin(26))

while True:
  print(adc.read_u16())
  time.sleep(1)
```

Turn the potentiometer to see your maximum and minimum values. They should be approximately between 0 and 65025. You can now use this value to control the duty cycle for PWM on the LED. Change the code to the following. Once you have run it, tune the dial on the potentiometer to control the LED's brightness.

```
from machine import Pin, PWM, ADC

pwm = PWM(Pin(15))
adc = ADC(Pin(26))

pwm.freq(1000)

while True:
  duty = adc.read_u16()
  pwm.duty_u16(duty)
```

Power your Pico
If you want to run your Pico without it being attached to a computer, you need to use a USB power supply. Safe operating voltages are between 1.8V and 5.5V.
To automatically run a MicroPython program, simply save it to the device with the name `main.py`. In Thonny, click on the File menu and then Save as for the last program you wrote.
When prompted, select '`MicroPython device`' from the pop-up menu.
Name your file `main.py`

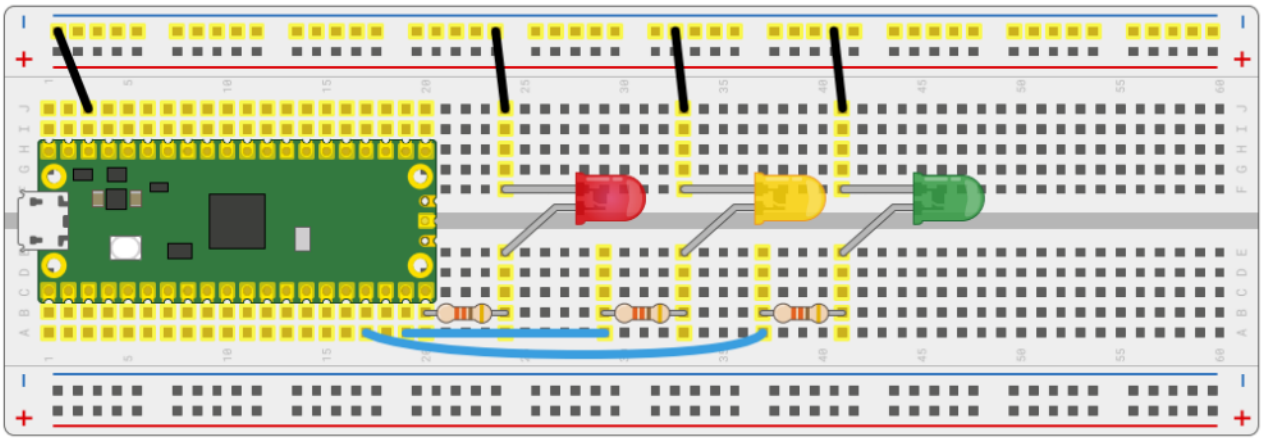
You can now disconnect your Pico from your computer and use a micro USB cable to connect it to a mobile power source, such as a battery pack.
Once connected, the main.py file should run automatically so you can interact with the components attached to your Pico.

One more project: Traffic light controller

For this project, you'll need a red, a yellow or amber, and a green LED and three 330 Ω resistors. Later on also a pushbutton and an active piezoelectric buzzer.

The simple one

Start by building a simple traffic light system, as shown below.



Create a new program, and start by importing the `machine` library so you can control your Pico's GPIO pins:

```
import machine
```

You'll also need to import the `utime` library, so you can add delays between the lights going on and off

```
import utime
```

As with any program using your Pico's GPIO pins, you'll need to set each pin up before you can control it

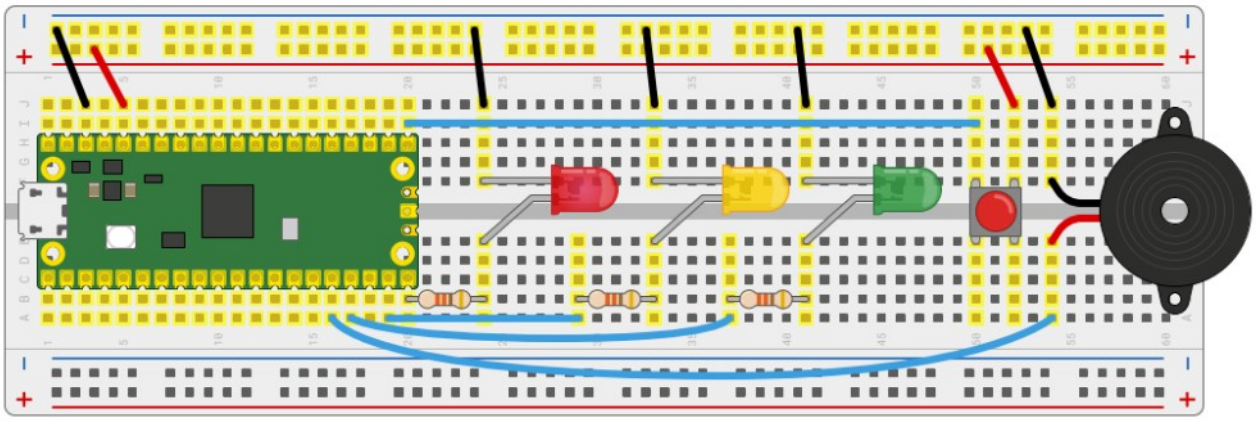
```
led_red   = machine.Pin(15, machine.Pin.OUT)
led_amber = machine.Pin(14, machine.Pin.OUT)
led_green = machine.Pin(13, machine.Pin.OUT)
```

These lines set pins GP15, GP14, and GP13 up as outputs, and each is given a descriptive name to make it easier to read the code.
Real traffic lights don't run through once and stop – they keep going, even when there's no traffic there and everyone's asleep. So that your program does the same, you'll need to set up an infinite loop:

```
while True:
    led_red.value(1)
    utime.sleep(5)
    led_red.value(0)
    led_green.value(1)
    utime.sleep(5)
    led_green.value(0)
    led_amber.value(1)
    utime.sleep(5)
    led_amber.value(0)
```

Click the Run icon and save your program to your Pico as `Traffic_Lights.py`. Watch the LEDs.

The more complex one
The pattern will loop until you press the Stop button, because it forms an infinite loop. It's based on the traffic light pattern used in real-world traffic control systems.
Real traffic lights aren't just there for road vehicles, though: they are also there to protect pedestrians, giving them an opportunity to cross a busy road safely.
To turn your traffic lights into a puffin crossing, you'll need two things: a push-button switch, so the pedestrian can ask the lights to let them cross the road and a buzzer, so the pedestrian knows when it's their turn to cross. Wire those into your breadboard as below, with the switch wired to pin GP16 and the 3V3 rail of your breadboard, and the buzzer wired to pin GP12 and the ground rail of your breadboard.



In Thonny, go back to the lines where you set up your LEDs and add the following two new lines below:

```
button = machine.Pin(16, machine.Pin.IN, machine.Pin.PULL_DOWN)
buzzer = machine.Pin(12, machine.Pin.OUT)
```

This sets the button on pin GP16 up as an input, and the buzzer on pin GP12 as an output. Remember, your Pico has built-in programmable resistors for its inputs, which run as pull-down resistors by default.
Next, you need a way for your program to constantly monitor the value of the button. Previously, all your programs have worked step-by-step through a list of instructions – only ever doing one thing at a time. Your traffic light program is no different: as it runs, MicroPython walks through your instructions step-by-step, turning the LEDs on and off.
For a basic set of traffic lights, that's enough. For a puffin crossing, though, your program needs to be able to record whether the button has been pressed in a way that doesn't interrupt the traffic lights. To make that work, you'll need a new library: `_thread`.
Go back to the section of your program where you import the machine and utime libraries, and import the `_thread` library

```
import _thread
```

A thread or thread of execution is, effectively, a small and partially independent program. You can think of the loop you wrote earlier, which controls the lights, as the main thread of your program – and using the `_thread` library you can create an additional thread, running at the same time.
At the moment, your program has only one thread – the one which controls the traffic lights. The RP2040 microcontroller which powers your Pico, however, has two processing cores – meaning, you can run two threads at the same time to get more work done.
Before you can make another thread, you'll need a way for the new thread to pass information back to the main thread – and you can do this using global variables. The variables you've been working with prior to this are known as local variables, and only work in one section of your program; a global variable works everywhere, meaning one thread can change the value

and another can check to see if it has been changed.

To start, you need to create a global variable. Below `buzzer = machine.Pin(12, machine.Pin.OUT)`, add the following:

```
global button_pressed
button_pressed = False
```

This sets up `button_pressed` as a global variable, and gives it a default value of `False` – meaning when the program starts, the button hasn't yet been pushed. The next step is to define your thread, by adding the following lines directly below – adding a blank line, if you want, to make your program more readable

```
def button_reader_thread():
    global button_pressed
    while True:
        if button.value() == 1:
            button_pressed = True
```

The first line you've added defines your thread and gives it a name which describes its purpose: a thread to read the button input.
The next line lets MicroPython know you're going to be changing the value of the global `button_pressed` variable. If you only want to check the value, you wouldn't need this line – but without it you can't make any changes to the variable.
Next, you've set up a new loop. The next line is a conditional which checks to see if the value of the button is 1. Because your Pico uses an internal pull-down resistor, when the button isn't being pressed the value read is 0 – meaning the code under the conditional never runs. Only when the button is pressed will the final line of your thread run: a line which sets the `button_pressed` variable to True, letting the rest of your program know the button has been pushed.
You might notice there's nothing in the thread to reset the button_pressed variable back to False when the button is released after being pushed. There's a reason for that: while you can push the button of a puffin crossing at any time during the traffic light cycle, it only takes effect when the light has gone red and it's safe for you to cross. All your new thread needs to do is to change the variable when the button has been pushed; your main thread will handle resetting it back to `False` when the pedestrian has safely crossed the road.

Defining a thread doesn't set it running: it's possible to start a thread at any point in your program, and you'll need to specifically tell the `_thread` library when you want to launch the thread. Unlike running a normal line of code, running the thread doesn't stop the rest of the program: when the thread starts, MicroPython will carry on and run the next line of your program even as it runs the first line of your new thread.
Create a new line below your thread, deleting all of the indentation Thonny has automatically added for you, which reads:

```
_thread.start_new_thread(button_reader_thread, ())
```

This tells the `_thread` library to start the thread you defined earlier. At this point, the thread will start to run and quickly enter its loop – checking the button thousands of times a second to see if it's been pressed yet. The main thread, meanwhile, will carry on with the main part of your program. Click the `Run` button now. You'll see the traffic lights carry on their pattern exactly as before, with no delay or pauses. If you press the button, though, nothing will happen – because you haven't added the code to actually react to the button yet.

Go to the start of your main loop, directly underneath the line `while True:`, and add the following code

```
if button_pressed == True:
    led_red.value(1)
    for i in range(10):
        buzzer.value(1)
        utime.sleep(0.2)
        buzzer.value(0)
        utime.sleep(0.2)
    global button_pressed
    button_pressed = False
```

This chunk of code checks the `button_pressed` global variable to see if the push-button switch has been pressed at any time since the loop last ran. If it has, as reported by the button reading thread you made earlier, it begins running a section of code which starts by turning the red LED on to stop traffic and then beeps the buzzer ten times – letting the pedestrian know it's time to cross.
Finally, the last two lines reset the `button_pressed` variable back to `False` – so the next time the loop runs it won't trigger the pedestrian crossing code unless the button has been pushed again. You'll see you didn't need the line global `button_pressed` to check the status of the variable in the conditional; it's only needed when you want to change the variable and have that change affect other parts of your program.

Your finished program should look like this:

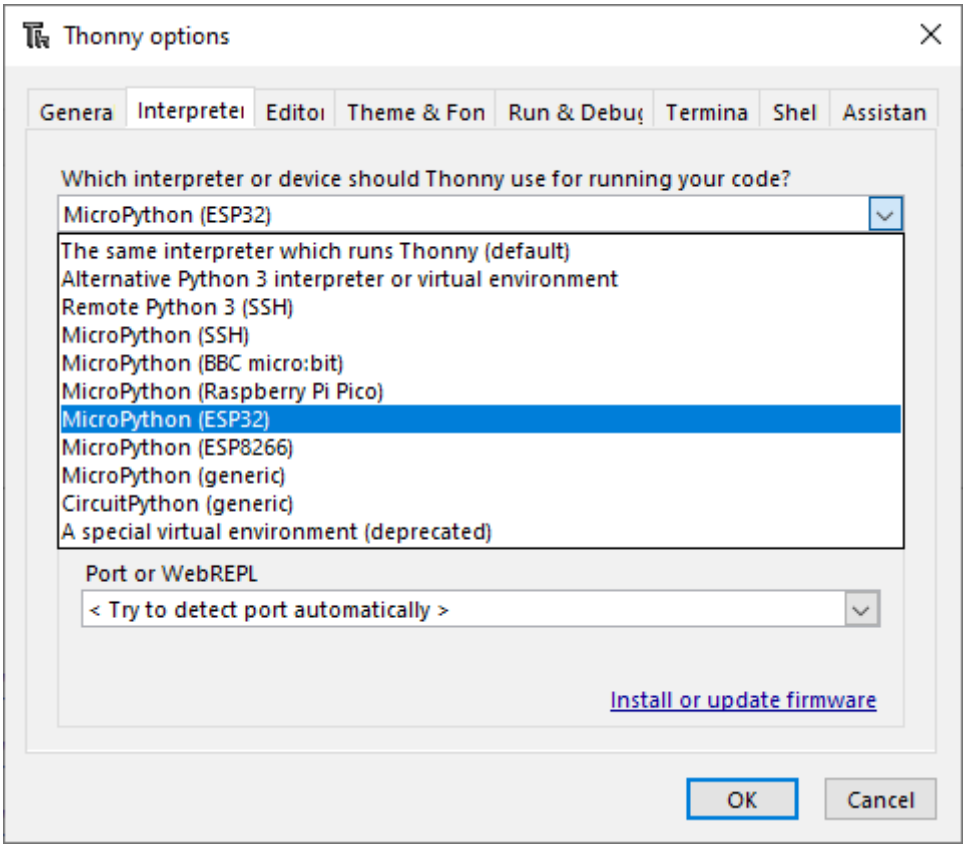
```
import machine
import utime
import _thread

led_red   = machine.Pin(15, machine.Pin.OUT)
led_amber = machine.Pin(14, machine.Pin.OUT)
led_green = machine.Pin(13, machine.Pin.OUT)
button    = machine.Pin(16, machine.Pin.IN, machine.Pin.PULL_DOWN)
buzzer    = machine.Pin(12, machine.Pin.OUT)

global button_pressed
button_pressed = False

def button_reader_thread():
   global button_pressed
   while True:
      if button.value() == 1:
          button_pressed = True

_thread.start_new_thread(button_reader_thread, ())

while True:
    if button_pressed == True:
        led_red.value(1)
        for i in range(10):
            buzzer.value(1)
            utime.sleep(0.2)
            buzzer.value(0)
            utime.sleep(0.2)
        global button_pressed
        button_pressed = False

   led_red.value(1)
   utime.sleep(5)
   led_red.value(0)
   led_green.value(1)
   utime.sleep(5)
   led_green.value(0)
   led_amber.value(1)
   utime.sleep(5)
   led_amber.value(0)
```

Click the `Run` icon. At first, the program will run as normal: the traffic lights will go on and off in the usual pattern. Press the push-button switch: if the program is currently in the middle of its loop, nothing will happen until it reaches the end and loops back around again – at which point the light will go red and the buzzer will beep to let you know it's safe to cross the road. The conditional section of code for crossing the road runs before the code you wrote earlier for turning the lights on and off in a cyclic pattern: after it's finished, the pattern will begin as usual with the red LED staying lit for a further five seconds on top of the time it was lit while the buzzer was going. This mimics how a real puffin crossing works: the red light remains lit even after the buzzer has stopped sounding, so anyone who started to cross the road while the buzzer was going has time to reach the other side before the traffic is allowed to go.
Let the traffic lights loop through their cycle a few more times, then press the button again to trigger another crossing.

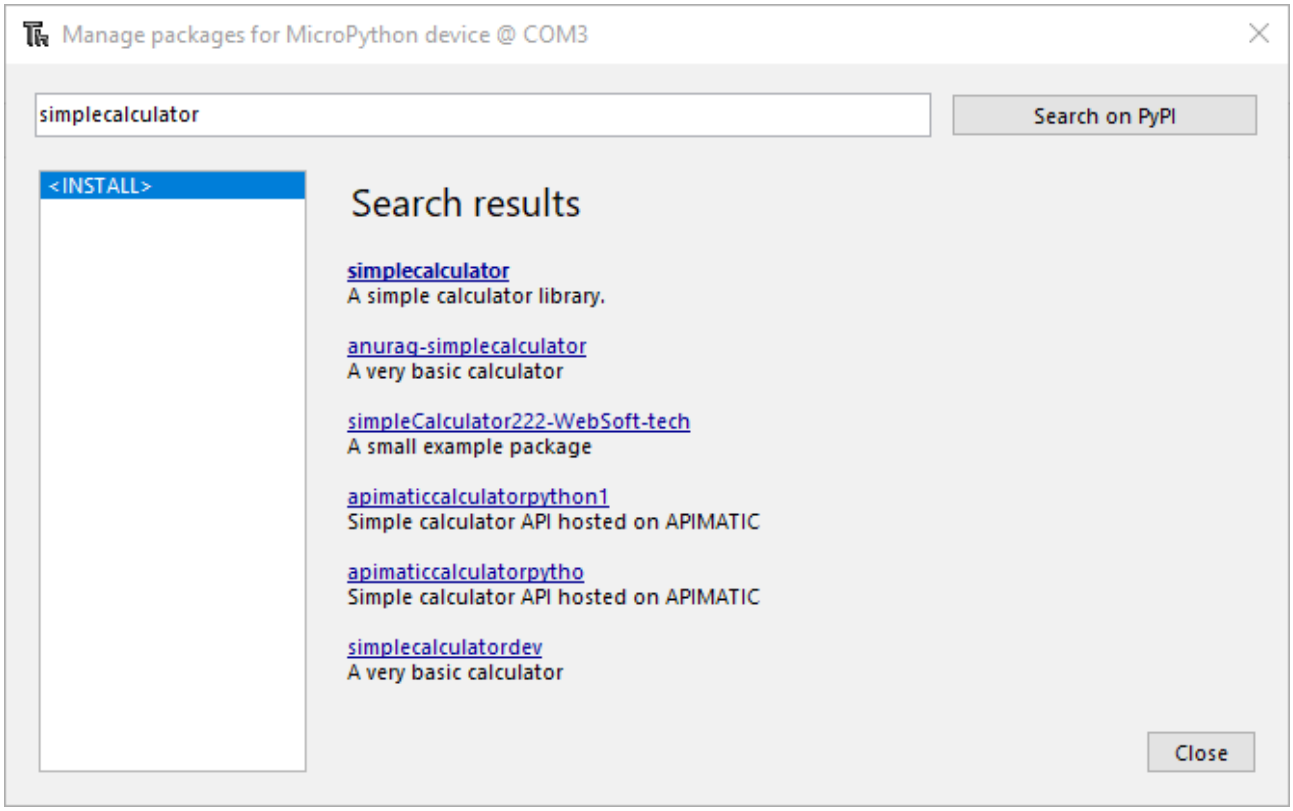
More on Thonny IDE

Debug Your Code

Debugger is only available when you have selected 'The same interpreter which runs Thonny (default)' in Tools → Options → Interpreter. If you select 'MicroPython (ESP32)', debugging is not available

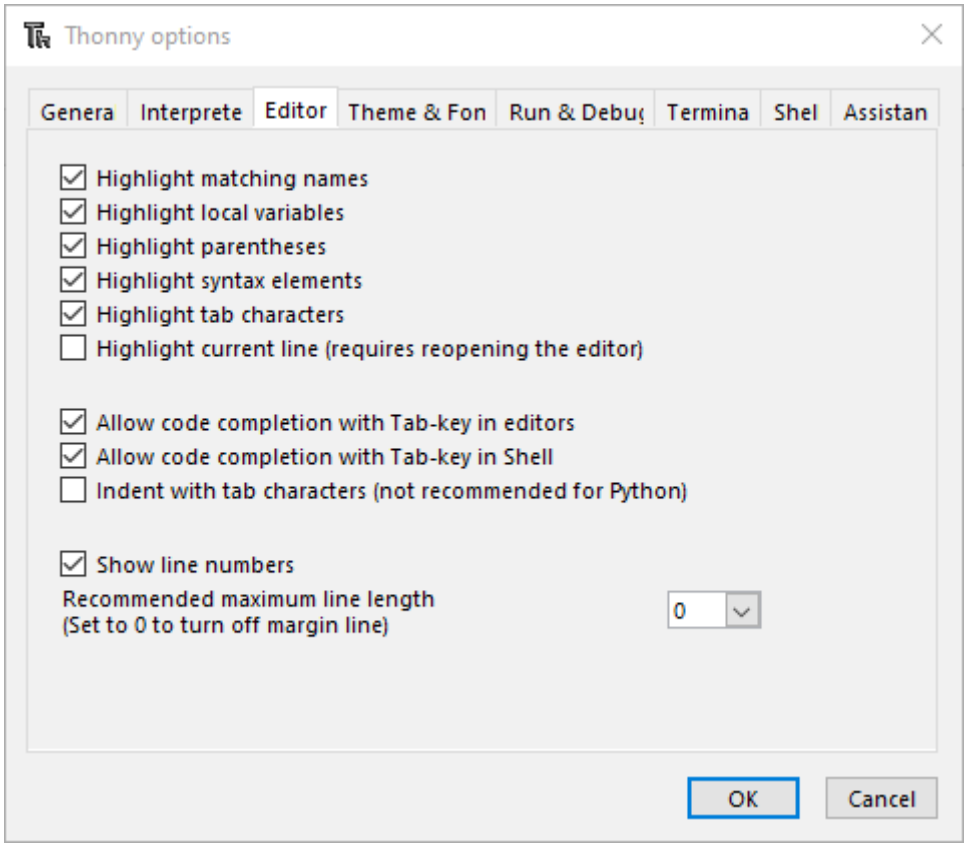


The Package Manager
As you continue to learn Python, it can be quite useful to download a Python package to use inside of your code. This allows you to use code that someone else has written inside of your program.
Consider an example where you want to do some calculations in your code. Instead of writing your own calculator, you might want to use a third-party package called `simplecalculator`. In order to do this, you'll use Thonny's package manager.
The package manager will allow you to install packages that you will need to use with your program. Specifically, it allows you to add more tools to your toolbox. Thonny has the built-in benefit of handling any conflicts with other Python interpreters.
To access the package manager, go to the menu bar and select `Tools > Manage Packages…` This should pop open a new window with a search field. Type `simplecalculator` into that field and click the `Search on PyPi` button.
The output should look similar to this:



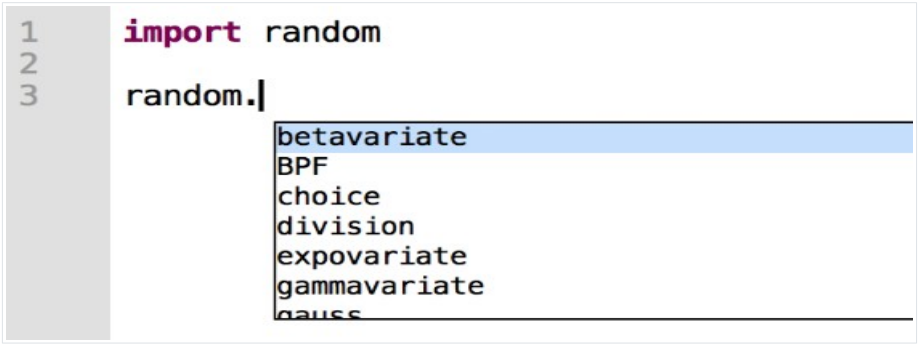
Select the right one (which is the first) and go ahead to click `Install` to install this package. You will see a small window pop up showing the system's logs while it installs the package. Once it completes, you are ready to use `simplecalculator` in your code.

Variable Scope Highlighting
Thonny offers variable highlighting to remind you that the same name doesn't always mean the same variable. In order for this feature to work, on the menu bar, go to `Tools → Options → Editor` and ensure that `Highlight matching names` is checked.



This feature can really help you avoid typos and understand the scope of your variables.

Code Completion
Thonny also offers code completion for APIs. Notice in the snapshot below how pressing the Tab key shows the methods available from the random library:



This can be very useful when you're working with libraries and don't want to look at the documentation to find a method or attribute name.

Troubleshooting Tips for Thonny IDE
We've discovered some common problems and error messages that occur with Thonny IDE. Usually restarting your ESP with the on-board EN/RST button fixes your problem. Or pressing the Thonny IDE *"Stop/Restart backend"* button and repeating your desired action. In case it doesn't work for you, read these next common errors and discover how to solve them.

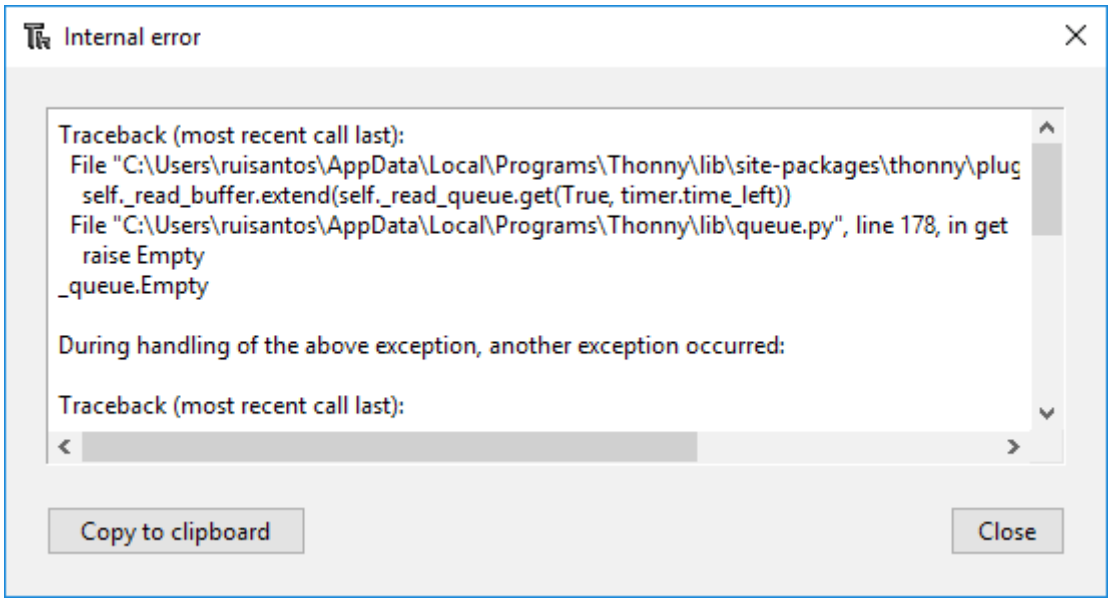
Error #1: You get one of the following messages:

```
========================= RESTART =========================
Unable to connect to COM4
Error: could not open port 'COM4': FileNotFoundError(2, 'The system cannot
find the file specified.', None, 2)

Check the configuration, select Run → Stop/Restart or press Ctrl+F2 to try
again. (On some occasions it helps to wait before trying again.)
Or:
========================= RESTART =========================
Could not connect to REPL.
Make sure your device has suitable firmware and is not in bootloader mode!
Disconnecting.
Or:
========================= RESTART =========================
Lost connection to the device (EOF).
```

Unplug, and plug back your ESP board. Then, double-check that you've selected the right serial port in the *Tools* > *Options* > *Interpreter* > *Port*. Then, click the *"Stop/Restart backend"* button to establish a serial communication. You should now be able to upload a new script or re-run new code.
These errors might also mean that you have your serial port being used in another program (like a serial terminal or in the Arduino IDE). Double-check that you've closed all the programs that might be establishing a serial communication with your ESP board. Then, unplug and plug back your ESP board. Finally, restart Thonny IDE.

Error #2: Thonny IDE fails to respond or gives an Internal Error

When this happens, you can usually close that window and it will continue to work. If it keeps crashing, we recommend restarting the Thonny IDE software.



Error #3: Thonny IDE hangs when pressing "**Stop/Restart backend**" button

When you press the "Stop/Restart backend" button, you need to wait a few seconds. The ESP needs time to restart and establish the serial communication with Thonny IDE. If you press the "Stop" button multiple times or if you press the button very quickly, the ESP will not have enough time to restart properly and it's very likely to crash Thonny IDE.

Error #4: Problem restarting your ESP board, running a new script or opening the serial port
Brownout detector was triggered
Or if the ESP keeps restarting and printing the ESP boot information:

```
ets Jun 8 2016 00:22:57

rst:0x1 (POWERON_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:2
load:0x3fff0018,len:4
load:0x3fff001c,len:4732
load:0x40078000,len:7496
load:0x40080400,len:5512
```

The "Brownout detector was triggered" error message or constant reboots means that there's some sort of hardware problem. It's often related to one of the following issues

- Poor quality USB cable;
- USB cable is too long;
- Board with some defect (bad solder joints);
- Bad computer USB port;
- Or not enough power provided by the computer USB port.

Try a different shorter USB cable (with data wires), try a different computer USB port or use a USB hub with an external power supply.
Important: if you keep having constant problems or weird error messages, we recommend re-flashing your ESP board with the latest version of MicroPython firmware: Flash/Upload MicroPython Firmware to ESP32.

Error #5: When I try to establish a serial communication with the ESP32 in Thonny IDE, I can't get it to connect.

We think this is what's happening: when you're running a script in your board, sometimes it's busy running that script and performing the tasks.
So, you need to try start the connection by pressing *"Stop/Restart backend"* button multiple times or restart the ESP to catch available to establish the serial communication.
Warning: don't press the *"Stop/Restart backend"* button multiple times very quickly. After pressing that button, you need to be patient and wait a few seconds for the command to run. If you're running a script that uses Wi-Fi, deep sleep, or it's doing multiple tasks, I recommend trying 3 or 4 times to establish the communication. If you can't, I recommend re-flash the ESP with MicroPython firmware.

Error #6: debug tools are grayed out: