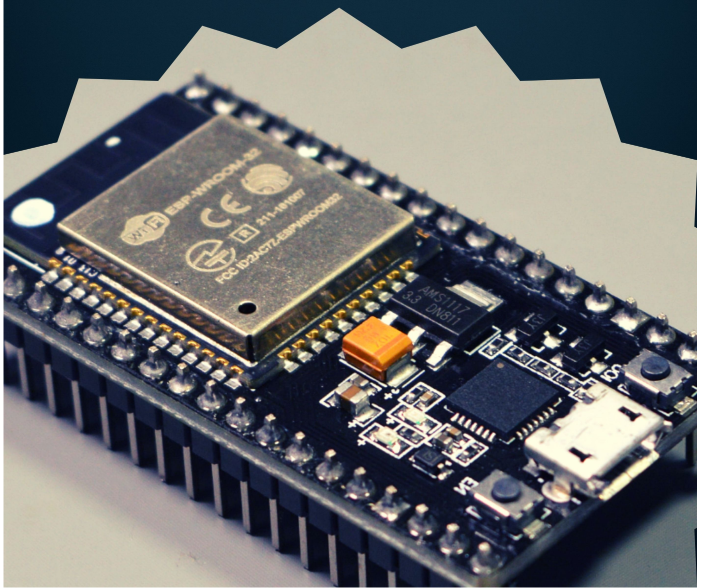
ESP32 MICROPYTHON

GET THE MOST OUT OF YOUR ESP32 WITH ARTICLES FROM THE TECH EXPLORATIONS BLOG



Peter Dalmaris, PhD

ESP32 MicroPython

Get the most out of your ESP32 with articles from the Tech Explorations Blog

Welcome to this special collection of articles, meticulously curated from the Tech Explorations blog and guides. As a token of appreciation for joining our email list, we offer these documents for you to download at no cost. Our aim is to provide you with valuable insights and knowledge in a convenient format. You can read these PDFs on your device, or print.

Please note that these PDFs are derived from our blog posts and articles with limited editing. We prioritize updating content and ensuring all links are functional, striving to enhance quality continually. However, the editing level does not match the comprehensive standards applied to our Tech Explorations books and courses.

We regularly update these documents to include the latest content from our website, ensuring you have access to fresh and relevant information.

License statement for the PDF documents on this page

Permitted Use: This document is available for both educational and commercial purposes, subject to the terms and conditions outlined in this license statement.

Author and Ownership: The author of this work is Peter Dalmaris, and the owner of the Intellectual Property is Tech Explorations (https://techexplorations.com). All rights are reserved.

Credit Requirement: Any use of this document, whether in part or in full, for educational or commercial purposes, must include clear and visible credit to Peter Dalmaris as the author and Tech Explorations as the owner of the Intellectual Property. The credit must be displayed in any copies, distributions, or derivative works and must include a link to https://techexplorations.com.

Restrictions: This license does not grant permission to sell the document or any of its parts without explicit written consent from Peter Dalmaris and Tech Explorations. The document must not be modified, altered, or used in a way that suggests endorsement by the author or Tech Explorations without their explicit written consent.

Liability: The document is provided "as is," without warranty of any kind, express or implied. In no event shall the author or Tech Explorations be liable for any claim, damages, or other liability arising from the use of the document.

By using this document, you agree to abide by the terms of this license. Failure to comply with these terms may result in legal action and termination of the license granted herein.

1. Introduction to MicroPython with the ESP32: What is MicroPython?

MICROPYTHON WITH THE ESP32 GUIDE SERIES

Introduction To MicroPython With The ESP32: What Is MicroPython?

The Guides in this series are dedicated to MicroPython for the ESP32. In this first lesson, I will introduce you to MicroPython, its reason to exist, how it relates to Python, and its most important characteristics.

Around mid-2014, Damien George published a new programming language for microcontrollers, called MicroPython. This publication was a successful completion of

an ambitious Kickstarter project that begun in 2013.

At the time, microcontroller programming was dominated by the C language.

If you are familiar with the Arduino, you know what C looks like. On a microcontroller like the Arduino, C is not very difficult to learn, however things do get more complicated as programs get bigger.

As microcontrollers started becoming more powerful, more people started being interested in programming them. Many of them were first-time programmers. This included people in all age brackets.

Damien wanted to create a language that would work on microcontrollers that would be much easier to learn and use than C. He did not want to reinvent the wheel, so he chose Python as his prototype.

His challenge was to create a language that can mimic Python that can run on the bare metal of a microcontroller, without an operating system.

So, he created MicroPython.

What is MicroPython?

Here is a description of the language from the MicroPython website (emphasis in bold is mine):

"MicroPython is a **lean** and **efficient** implementation of the Python 3 programming language that includes a **small subset** of the Python standard library and is optimised to run on **microcontrollers** and in constrained environments."

Python MicroPython

Because MicroPython contains the word "Python", it is easy to become confused and think that MicroPython is simply a

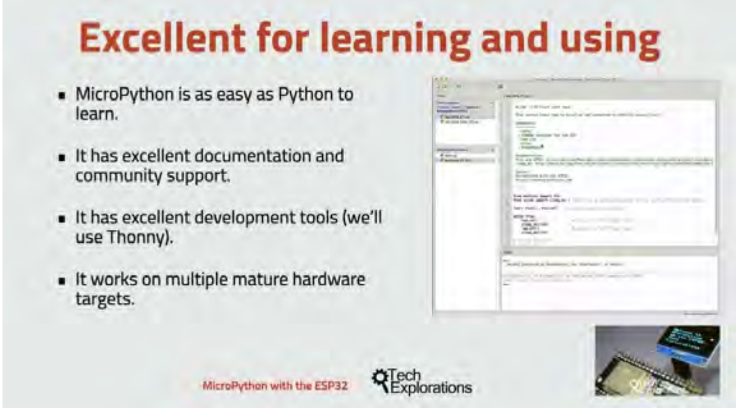
smaller version of Python.

It is the same confusion that I have seen in the past between Java and Javascript.

While Python and MicroPython have a similar name, they are totally different languages, with a different set of goals and implementation.

I talk more about the differences between Python and MicroPython in a later lecture. For now, I just want to make sure that you are not confused by the similarity in the name.

Excellent for learning and using



What MicroPython has taken from Python is the language architecture, its programming philosophy for code readability, and a huge pool of programmers that already know how to use Python.

Pythonistas can quickly become MicroPythonistas and write programs for microcontrollers.

According to the "PYPL PopularitY of Programming Language Index", Python is the most popular programming language in

the world, with a 30% share. This index is calculated based on the amount of searching is done on Google for programming language tutorials or resources.

As a comparison, C/C++ used by the Arduino boards ranks 5th place in this index.

This popularity translates to a Python universe that is filled with all the documentation, libraries and community support you will ever need.

MicroPython is as easy as Python to learn, and follows Python's tradition for excellent development tools and documentation. In this course, you will see me constantly browsing through the MicroPython core documentation, as well as many of the excellent libraries we'll be using.

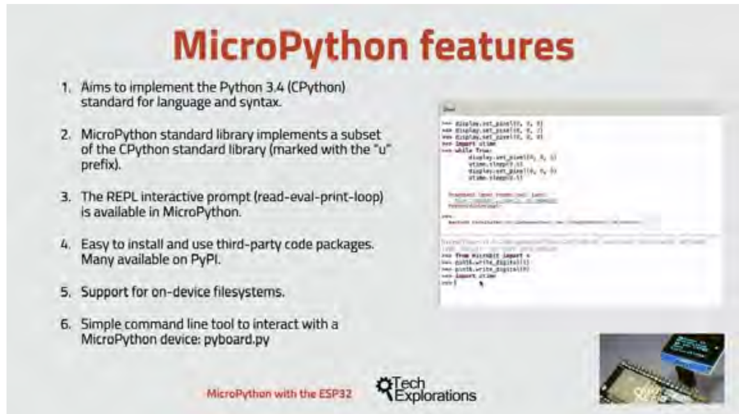
In terms of tools, you have many choices. In this course, I'll be using Thonny. But, you can also choose tools such as upycraft, and the Mu editor.

What I really like about Thonny is that is a full Python editor on its own merit, with excellent debugging tools, but also fully supports MicroPython on the ESP32 as well as other target boards like the Raspberry Pi Pico and the BBC Microbit.

Another big advantage of MicroPython is that once you learn it, you can use your skills across multiple hardware targets. At the time I am recording this lecture, MicroPython has support for the original Myboard v1 and D-series, as well as third-party boards such as the STM32 Nucleo and Discovery boards, the Espruino Pico, the Raspberry Pi Pico, the WiPy, the ESP8266 and ESP32, the TinyPico, and the BBC Micro:Bit.

This was just a partial list.

MicroPython features



Let's take a quick tour of MicroPython's most important features.

Implements Python 3.4 language and syntax

First and most important for anyone new to this language, is that MicroPython aims to implement the Python 3.4 standard for language and syntax.

This simply means that anyone who already programs in Python 3.4 will be able to start programming in MicroPython immediately. Python 3.4 reserved keywords, operators, functions, even the infamous whitespace indentation is faithfully implemented in MicroPython.

uPython standard library is a subset of CPython standard library

Because MicroPython targets embedded computers and microcontrollers, it is not possible to implement the full Python standard library, with all of its modules and methods.

There's simply not enough storage on the target devices.

Therefore, MicroPython implements a selected subset of CPython's standard library. Even that, is implemented with emphasis in efficiency. MicroPython versions of Python libraries have a name with the "u" letter prefix.

uPython has a REPL

MicroPython has an Interactive Interpreter mode, also known as "REPL". REPL stands for "Read-Eval-Print-Loop". Think of it as a command line for Python. You can use this command line to issue Python instructions or even code blocks. The REPL will evaluate this Python code immediately.

The MicroPython REPL is fully featured, with auto-indent, auto-completion, ability to interrupt a running program with Ctrl-C and invoke a soft reset.

There is also a paste mode, and you can use the special "_" underscore variable that stores the output of the previous computation. I this course, I'll be using the REPL extensively to demonstrate and test code.

Easy to install 3rd-party packages

Outside of the MicroPython standard library, there are countless libraries contributed by users and published online on repositories like Github and PyPI (the Python Package Index).

Similar to CPython, MicroPython has a simple mechanism for including external code to your programs. In this course, I'll show you how to find and use external libraries that make it easy to integrate hardware components like screen and sensors to your MicroPython projects.

Supports on-device filesystems

MicroPython has the ability to access a small filesystem on the target miroctroller device. This filesystem makes it possible to

store your MicroPython programs, supporting library files, and arbitrary files such as text files for storing sensor data or credentials for networks and IoT services, or bitmap image files.

In this course, I have prepared several examples where I demonstrate how to use the file system on the ESP32.

Simple command-line tool

MicroPython has a single command-line Python tool that allows you to run a script or access the filesystem on a target device.

This tool is called pyboard.py.

In this course, we will not be using this tool because Thonny IDE has build-in support for MicroPython on a variety of target devices, including the ESP32. However, I mention pyboard.py here because it provides a simple way to interact with the REPL, run programs, and upload or download files from the device file system.

MicroPython on different devices

MicroPython works on many different microcontrollers.

The diversity of the hardware means that not all MicroPython code will work across those devices without modifications.

In general, there are two points to remember in relation to sharing MicroPython code across different targets.

uPython language and core libraries work across different targets

Most of the code that uses MicroPython standard library functions and the core of the language will work without modifications.

Language syntax, reserved keywords, control structures, and functions that come from standard libraries like math (for mathematics), uos (for basic operating system services) and utime (for time and date related functions), will work across all MicroPython hardware targets.

Code that controls pins, interfaces needs customisation

On the other hand, any functionality that is uniquely implemented on a microcontroller requires a unique implementation in MicroPython.

For example, the way that digital pin functionality is implemented between the ESP32 and the Raspberry Pi Pico differs.

It is it a similar case to how functions relating to network, the I2C and SPI interfaces, and the analog to digital converters are implemented across boards. These differences are reflected in the MicroPython implementation for each board.

For this reason, in addition to the standard library, MicroPython has libraries specifically implemented for each supported board. You should take a bit of time to study your target

device special libraries so that you know what is available and how you can go about taking advantage of the device capabilities.

uPython may not access all hardware features

One more thing: Not al device capabilities can be access through MicroPython. For example, in the ESP32, there is no MicroPython module for Bluetooth, although there is for Wifi.

MicroPython is readable

Let’s wrap up this lesson by going back to MicroPython’s most important attribute.

MicroPython, like CPython, is designed to be readable. It almost reads like natural language.

This is an example code segment from one of the lectures in this course. Even if you have never seen MicroPython before, and perhaps have never programmed before, you will be able to make fairly accurate inferences about what this code is supposed to do.

Yes, you do need to have a basic understanding of electronics. Without that, keywords like “Pin.OUT” and “Pin.PULL_UP” will not make sense.

However, the language barrier to entry for MicroPython is minimal. Certainly, it is much lower than the barrier to entry for a language like C or C++.

This is the number 1 reason why Python became so popular, and why MicroPython has been gaining massive support in popularity since the Kick-starter campaign in 2014.

Learn MicroPython for the ESP32

With this video course, you will learn how to use theMicroPython programming language with the ESP32 micro-controller.

MicroPython is the perfect language for anyone looking for the easiest (yet still powerful) way to program a micro-controller.

__CONFIG_colors_palette__{“active_palette”:0,”config”:{“colors”:{“3e1f8”:{“name”:”Main Accent”,”parent”:-1}},”gradients”:[]},”palettes”:[{“name”:”Default Palette”,”value”:{“colors”:{“3e1f8”:{“val”:”rgb(217, 49, 33)”}},”gradients”:[]},”original”:{“colors”:{“3e1f8”:{“val”:”rgb(19, 114, 211)”,”hsl”:{“h”:210,”s”:0.83,”l”:0.45}}},”gradients”:[]}}]}__CONFIG_colors_palette__

2. MicroPython vs CPython

MICROPYTHON WITH THE ESP32 GUIDE SERIES

MicroPython Vs CPython

In this lesson, I will discuss some of the differences between MicroPython and CPython that I believe are most important to know when you are getting started with MicroPython.

As you know by now, MicroPython and CPython are two different programming languages. MicroPython has copied CPython as faithfully as possible to create a high-level language programming experience for microcontrollers. There are differences between the two languages, which I would like to summarize in the next few minutes.

See the MicroPython documentation for detailed differences between MicroPython and CPython

The differences between MicroPython and Python are documented in detail on the MicroPython website. There, you can find a complete list of those differences and code examples that demonstrate them. In this lesson, I will discuss some of the differences between MicroPython and CPython that I believe are most important to know when you are getting started with MicroPython.

Syntax

Let’s begin with syntax.

In CPython, you can do things like forget to put a space between a literal number and a keyword to form an expression, and that will be okay. CPython has enough flexibility to forgive mistakes like this. The same error in MicroPython, however, will generate a syntax error. MicroPython developers had to throw away the logic needed to deal with typos like that to fit the limited storage of the target devices.

Functions and “self”

Another example of these differences between the two languages is how the “self” keyword is handled. When self is used in a function in CPython, it does not count as an additional argument, but MicroPython does. As a result, if you provide an incorrect number of arguments to a function that contains the self keyword, then the error message it will get in CPython will be different from what you’ll get in MicroPython.

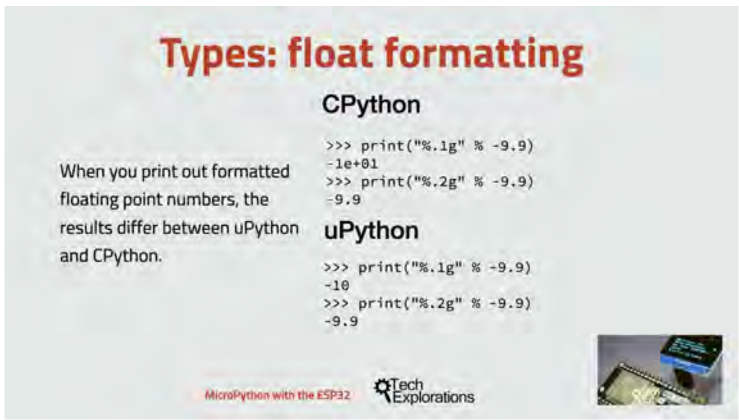
I’ve got an example here.

As you can see in this example, I’m calling the same function calculator in CPython and Python and I’m passing a single

argument. I should have passed two arguments, but I made a mistake. I just passed a single argument here. As you can see, I'm calling the same function with the same parameter. But the messages that are coming back to indicate the error are different. CPython is telling me that I've got one required argument missing. Where uPython is telling me that in total, there are supposed to be three arguments. I've only given two when, in fact, I've given one.

You can see that the message there is coming through here in MicroPython is or can be a bit confusing, which can throw you off and cause you to rely on your program's debugging. But as long as you are aware of the situation with the self keyword, I think you'll be able to get past issues like this.

Number formatting

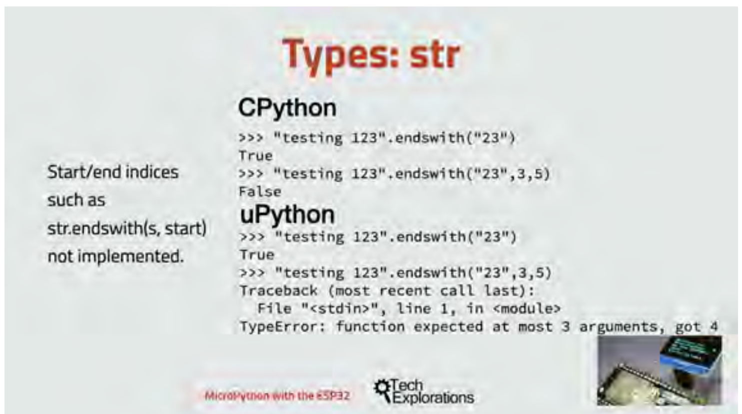


Here's another subtle difference that has to do with formatting in this particular example of a floating-point number.

When you print out formatted floating-point numbers, the result may differ between MicroPython and CPython. Here, I'm printing out this floating number, using the same commands between CPython and C Python – UPython being MicroPython. And as you can see that what comes out is different on each

occasion. In this case, when used with CPython, the “g” operator applies the exponential format to a number. In the output, the “g” operator differs between the two implementations.

Strings



Following is an example of differences relating to the string. As you may know, CPython contains powerful string manipulation functions, and not all of them are available on MicroPython. Two examples: “start with” and “end with”. These allow you to check if a string starts or ends with a specific character or string of characters.

In MicroPython, these functions only work in their basic format without the start and end index parameters. In the example here in this slide, you can see that the call to the end with function fails when we use it with the three parameters but work with a single parameter. In CPython, no problem. Either one will work correctly and as expected.

JSON

Next up, we've got JSON.

In this guide and course, we'll use the MicroPython JSON module to work with Internet of Things services. Unlike CPython and the CPython version of the JSON module, uJSON does not throw an exception if an object is not serializable.

And this means that if your program receives a JSON document from a web service that is not valid, you will not be able to deal with it gracefully using an exception handler. You will have to deal with this manually, or your program will crash.

Conclusion

All right, so these were just some of the subtle differences between CPython and the MicroPython implementation. And, of course, there are many more. Again, the best place to learn about them and keep track of the changes is the MicroPython documentation.

In the following lesson, I'll show you some of the best online resources for MicroPython. And these are the resources that you'll want to bookmark so that you can access them easily any time you work with MicroPython.

Learn MicroPython for the ESP32

With this video course, you will learn how to use theMicroPython programming language with the ESP32 micro-controller.

MicroPython is the perfect language for anyone looking for the easiest (yet still powerful) way to program a micro-controller.

__CONFIG_colors_palette__{“active_palette”:0,”config”:{“colors”:{“3e1f8”:{“name”:”Main Accent”,”parent”:-1}},”gradients”:[]},”palettes”:[{“name”:”Default Palette”,”value”:{“colors”:{“3e1f8”:{“val”:”rgb(217, 49, 33)”}},”gradients”:[]},”original”:{“colors”:{“3e1f8”:{“val”:”rgb(19, 114, 211)”,”hsl”:{“h”:210,”s”:0.83,”l”:0.45}}},”gradients”:[]}}]}__CONFIG_colors_palette__

MICROPYTHON WITH THE ESP32 GUIDE SERIES

MicroPython Resources

In this lesson, I will discuss MicroPython Resources that will save you time and help you to learn faster.

MicroPython is relatively new. Nevertheless, continuing that tradition set by Python is very well documented. In this lecture, I'll show you some of the best online, free, and community-supported resources that I use. I guarantee that these resources will save you time and help you take your first steps with MicroPython.

MicroPython.org

MicroPython.org is the home of the MicroPython language on the web; this is where you can find MicroPython firmware for the supported boards, links to the documentation, a discussion forum, and a store from where you can purchase “Pi” boards. I have prepared a separate lesson where I discuss supported hardware, so hold on to any hardware-related questions for now.

MicroPython documentation

In this course, we’ll spend a lot of time browsing the documentation, and I’ll talk about that shortly. I encourage you

to sign up for an account for the MicroPython forum, where you can participate in relevant discussions. The forums, including the one dedicated to the ESP32, are very busy with multiple new discussion threads almost daily.

We'll be spending a lot of time browsing through the MicroPython documentation. The documentation website is hosted under docs.micropython.org. It contains details about the MicroPython libraries, the language, and implementation. In almost every case, the documentation provides a detailed definition of every function and class and simple examples of how to use it. The documentation covers the Python standard libraries and MicroPython specific libraries and libraries specific to the "Pi" board, WiPy, ESP8266, and ESP32 boards.

MicroPython language reference

The MicroPython documentation focuses on topics specific to MicroPython. Because the MicroPython syntax, language, and programming philosophy comes from Python, you'll need to refer to the Python documentation from time to time.

For example, if you don't remember how to initialize a table, you can quickly look it up in the Python Documentation. You'll find this at docs.python.org/3 and then click on the Language

Reference link.

Python Package Index

MicroPython, as with Python, has a substantial library of packages created by its community of programmers.

A repository where you can find many of those packages is the Python Package Index at pypi.org. The Python Package Index contains packages designed for CPython and MicroPython.

You need to be a little careful when you search. Often packages written in MicroPython indicate that in the title.

For example, You can find a MicroPython package for the DHT12 sensor by searching for MicroPython DHT12 to distinguish it from other packages written for different platforms. One of them is PI GPIO, a Python package that works on the Raspberry Pi computer.

Of course, even when you find a package that explicitly indicates it is written for MicroPython, you need to check that it supports your hardware target. Not all of them do.

Awesome MicroPython

Awesome-MicroPython is a curated list of libraries for MicroPython specifically, that's unlike the Python Package Index.

In most cases, when I'm hunting for a MicroPython library, I go to Awesome MicroPython first. The curated list contains libraries grouped according to their purpose. You'll find libraries for Ethernet and MQTT communications, displays like E-Paper and LCD; there's GPIO and input/output libraries, all kinds of sensors, schedulers, storage, and much more.

But as with the Python Package Index, once you find a library that looks promising, you need to take a closer look and ensure that it will work with your microcontroller of choice. This information is not always readily available in the library's documentation. In many cases, you'll have to download the library and test it on your device to make sure that it's compatible with it.

Learn MicroPython for the ESP32

With this video course, you will learn how to use theMicroPython programming language with the ESP32 micro-

controller.

MicroPython is the perfect language for anyone looking for the easiest (yet still powerful) way to program a micro-controller.

__CONFIG_colors_palette__{“active_palette”:0,”config”:{“colors”:{“3e1f8”:{“name”:”Main Accent”,”parent”:-1}},”gradients”:[]},”palettes”:[{“name”:”Default Palette”,”value”:{“colors”:{“3e1f8”:{“val”:”rgb(217, 49, 33)”}},”gradients”:[]},”original”:{“colors”:{“3e1f8”:{“val”:”rgb(19, 114, 211)”,”hsl”:{“h”:210,”s”:0.83,”l”:0.45}}},”gradients”:[]}}]}__CONFIG_colors_palette__

4. MicroPython compatible boards

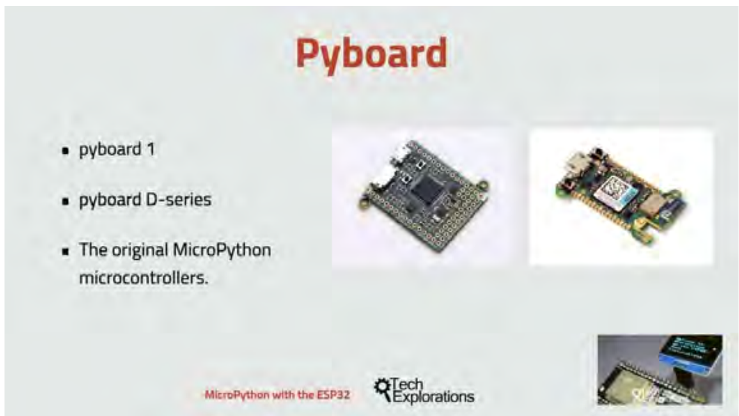
MICROPYTHON WITH THE ESP32 GUIDE SERIES

MicroPython Compatible Boards

In this lesson, I will discuss some of the microcontroller boards that are compatible with MicroPython. As you'll see, there's a huge variety to choose from.

When MicroPython was first published in 2014, only one board supported it, the original pyboard. A few years later, there's MicroPython support for a wide range of microcontrollers, including the ESP32, which is the one that we'll be using in this course.In this lesson, we'll take a closer look at the boards that can use MicroPython.

pyboard



Let’s start with the original pyboard. The pyboard 1 is the board that Damien George designed to run MicroPython for his Kickstarter project in 2014. The pyboard contains an STM32 microcontroller chip, which is based on an Cortex-M4CPU. It has 1024 kilobytes of flash ROM and 192 kilobytes of RAM. It also features a micro SD card slot for an expanded file system, and accelerometer, real time clock, four programable LEDS, 29 GPIOs, and two digital to analog converters among other things.

The new D-series pyboard also uses an STM32 microcontroller, but has a deep style form factor that makes it easier to integrate into projects. Got more flash and RAM capability for external flash, as well as Wi-Fi and Bluetooth connectivity, and improvements across the board. The pyboard is the golden standard for what a MicroPython device looks like.

ESP32 and ESP8266

Espressif ESP-based boards

- ESP8266
- ESP32
- Powerful, low cost, full-featured.

MicroPython with the ESP32 Tech Explorations

Then, of course, we have the Espressif, ESP, family of devices, the ESP32 and the older ESP8266 are almost fully supported by MicroPython. You have learned about the lack of Bluetooth support, for example, for the ESP32 in the previous lecture. The ESP32 specific libraries are documented on the main MicroPython documentation website.

Next to the pyboard, the ESP32 and ESP8266 seem to have the widest range of community contributed Micropython libraries. This means that there is a good chance that you'll be able to find a device driver for your favorite display or sensor. At the time I'm writing this, a Bluetooth is not supported and this is because of how much memory this implementation would require. Wi-Fi, however, as you probably already know, is fully functional.

So, apart from Bluetooth, almost all of the end user features on the ESP32 can be used in MicroPython, timers, GPIOs, PWM, Wi-Fi, I2C, SPI, sleep, analog to digital converters, all of those work. It's even possible to read the internal whole temperature sensors.

The ESP32 is the microcontroller that I've chosen to use in this course because of the excellent MicroPython implementation, the richness of its hardware, and my familiarity with it from previous projects.

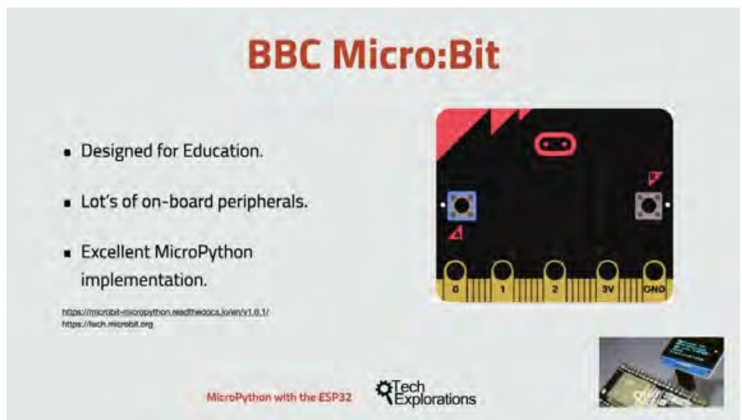
Raspberry Pi Pico

Now, let's have a look at the Raspberry Pi Pico. The Raspberry Pi Pico was released earlier in 2021 selling for around $5. And it's powered by the brand new RP2040 microcontroller. This microcontroller was designed by the Raspberry Pi Foundation. And very quickly, several new boards came out that are based on the same microcontroller, like the Feather 2040, the Tiny 2040. All of them can run the MicroPython firmware. And the Raspberry Pi Foundation provides excellent documentation through its website.

I find that compared to the pyboard and the ESP boards, it is much harder to find MicroPython device drivers for the Raspberry Pi Pico. It's still a new board, so I expect that this is going to change.

The Raspberry Pi Pico is an excellent, simple board. It doesn't have any wireless communications capability, but I think that this is a case where simplicity is an advantage. Along with the BBC micro:bit, the Raspberry Pi Pico is probably the easiest way to learn MicroPython.

BBC Micro:bit



Next up, the BBC micro:bit. So, the BBC micro:bit is a small board designed specifically for education.

It uses a Nordic nRF52833 application processor. And it contains an impressive array of built-in peripherals, such as an LED matrix display, a touch sensor, a microphone, a couple of buttons, and an accelerometer.

It also has a 2.4 gigahertz transceiver that students can experiment with and create a simple radio communications protocol and get microbits to talk to each other wirelessly.

The MicroPython implementation on the micro:bit is excellent as expected. I've tested many of its hardware components and everything seems to be working, even the radio communications.

STM32 boards

Next up, we've got the STM32 boards.

The Texas Instruments' Nucleo and Discovery boards and the Espruino Pico are based on the microcontrollers from the same STM32 family. I remind you that the pyboard also uses an STM32 microcontroller unit.

There are several Nucleo and Discovery boards geared towards rapid prototype development for engineers, but are also used in education. The Espruino Pico is a particularly popular board among makers because of how much power is packed in such a tiny board.

On the MicroPython website, it's mentioned that the STM32 line of microcontrollers from STM Microelectronics are officially supported by MicroPython via the STM32Cube HAL libraries. The STM32 port of MicroPython contains the source code for these MCUs.

This was just a short list of some examples of the boards that can work with MicroPython. In this course, we'll experiment with the ESP32, as you know.

Learn MicroPython for the ESP32

With this video course, you will learn how to use theMicroPython programming language with the ESP32 micro-controller.

MicroPython is the perfect language for anyone looking for the easiest (yet still powerful) way to program a micro-controller.

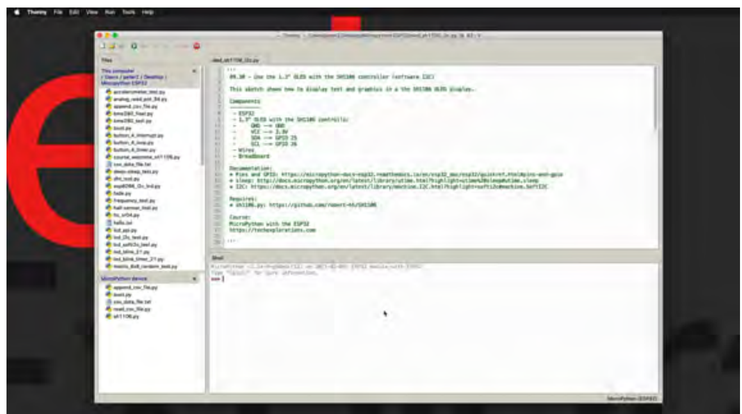
__CONFIG_colors_palette__{“active_palette”:0,”config”:{“colors”:{“3e1f8”:{“name”:”Main Accent”,”parent”:-1}},”gradients”:[]},”palettes”:[{“name”:”Default Palette”,”value”:{“colors”:{“3e1f8”:{“val”:”rgb(217, 49, 33)”}},”gradients”:[]},”original”:{“colors”:{“3e1f8”:{“val”:”rgb(19, 114, 211)”,”hsl”:{“h”:210,”s”:0.83,”l”:0.45}}},”gradients”:[]}}]}__CONFIG_colors_palette__

5. Getting started with Thonny IDE

MICROPYTHON WITH THE ESP32 GUIDE SERIES

Getting Started With The Thonny IDE

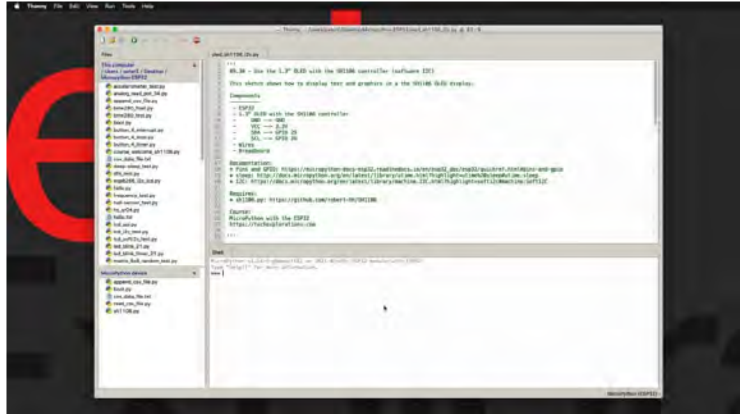
In this lesson, I will discuss the Thonny IDE, an open-source integrated development environment that we'll be using to program the ESP32 using MicroPython.



In this lesson I will show you around Thonny in my already set up instance, and show you the location where you can download the installation utility so that you can install it on your own computer.

To get the most out of this lesson, I recommend that you watch the video (see above).

Thonny IDE walkaround



Thonny IDE running on Mac OS

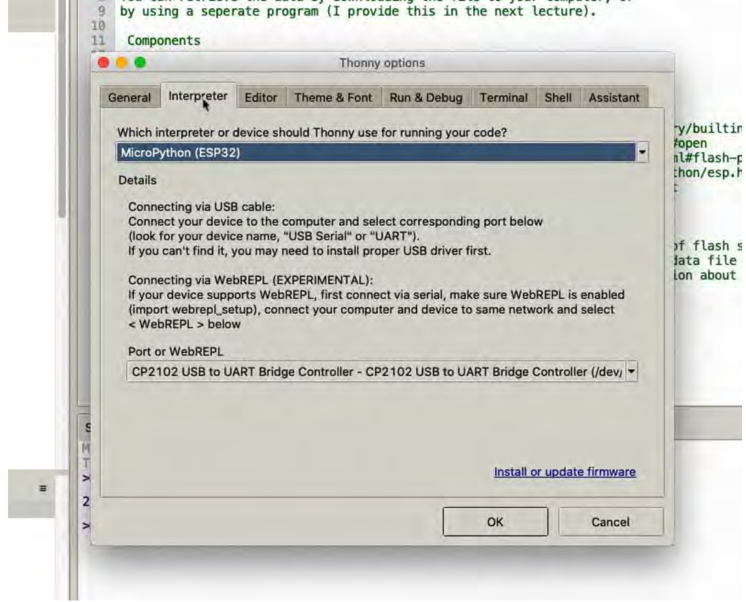
In the screenshot above you can see Thonny IDE running on Mac OS. I've done a little bit of configuration to customize the font types, and sizes, and things like that.

But, essentially, what you see here is Thonny as it looks like as soon as you install it.

Thonny is a competent and configurable integrated development environment. It would look like this at its most basic view, where you get the upper part of the window where you can see one or more tabs. You can have multiple tabs with your various Python programs or components for the program. And then, down below, you've got the Shell that you can use to interact with the Python interpreter.

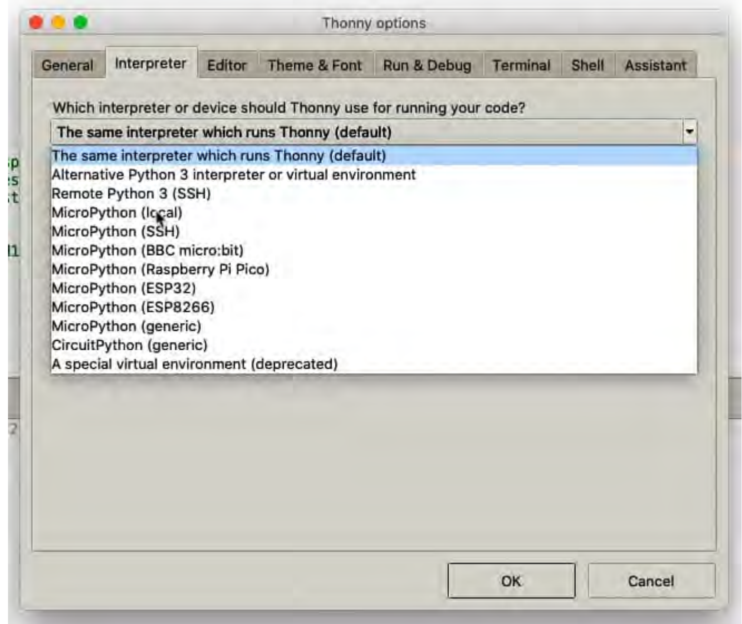
In this case, as you can see, I'm running MicroPython on my ESP32, which is connected. Don't worry about this for now. I will show you first how to install the necessary interpreter on your ESP32 in the following lesson. And then, show you how to make the connections and interact with MicroPython on the ESP32.

For now, all I want to show you is that the Shell allows me real-time interaction with the Python interpreter running on the ESP32. But apart from that, it’s got many more capabilities.



Thonny IDE works with multiple interpreters.

For example, if I go into Tools and Options, I can change the interpreter from MicroPython to one of the other available interpreters. For example, this one here is Python that ships with Thonny. Or you can go for Python that is running on a virtual environment or with Python running somewhere else. You may also access interpreters via a network, via SSH.



A list of interpreters that ship with Thonny IDE version 3.3.4 or later.

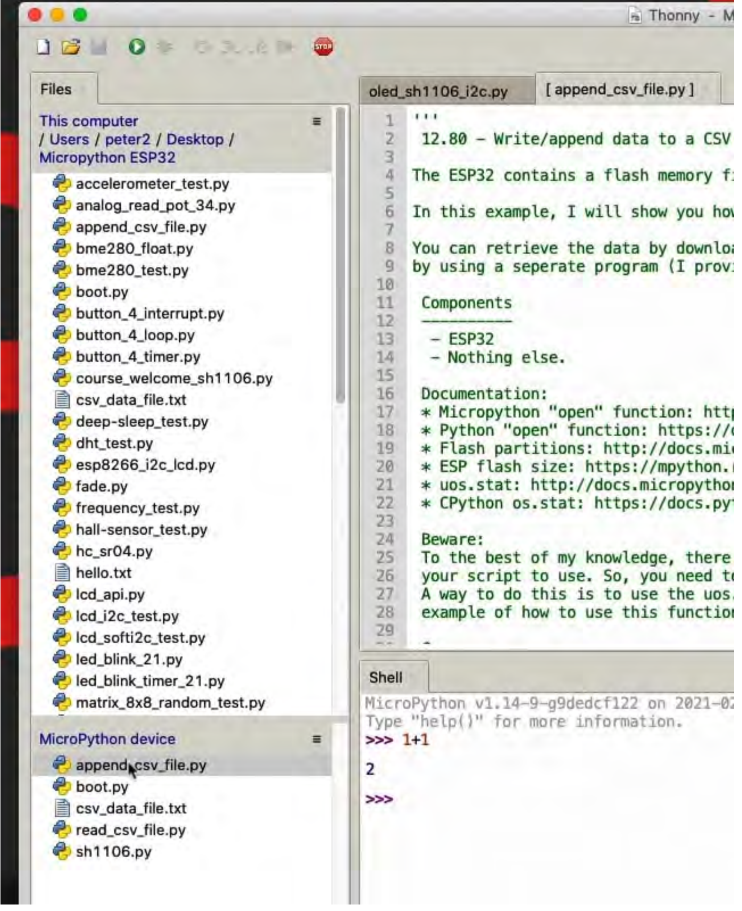
You can see here that the Thonny (version 3.3.4- latest at the time of recording) instance that I'm running ships with the capability of running MicroPython on BBC Micro:bit, Raspberry Pi Pico, ESP32, and the ESP8266; this is also a circuit Python environment. So, it's already fully featured just out of the box.

But you can install a lot more Python targets, as you can see, via plugins.

Thonny and Python interpreters

Another thing that I want to show you is that Thonny is used, not just for MicroPython on a microcontroller device, but for general Python development. And it gives you a lot of tools here, as you can see, to help you with that.

For example, you can turn on the files view, which gives you access to all files in a particular location on your local file system. In this case, it's on My Computer. It also gives you a view of the files that exist on the target device file system like these, so these files are stored on the ESP32 itself.



The file browser allows you to access files in the host computer and target device.

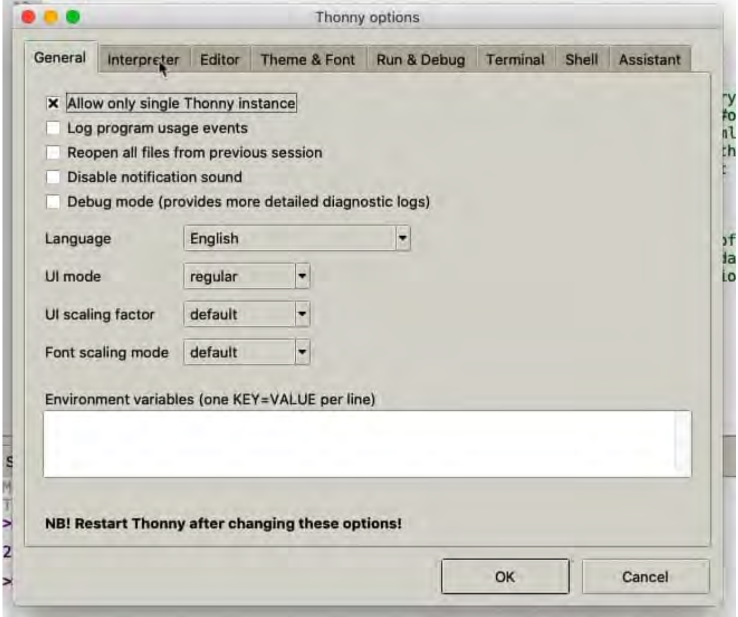
There's also a series of other types of tools, such as the ability to inspect the heap memory contents or, let's, say the stack,

which is useful when you are jumping from one function into another, keep track of which function you are in. I'm going to give you a little demonstration of this a little later in another section. You can check out the variables that have been set up and so on.

Thonny IDE provides many development tools, such as the variables, heap and stack inspectors.

Thonny IDE, important features

Let’s have a look at some of the most important features of Thonny. First of all, you’ve got the configuration window. You can access it from Preferences, but you can access the exact same thing by going to Tools and Options. And that allows you to customize the look and feel of Thonny, which forms you’re using, et cetera, how the debugger works, which terminal to use or Shell, and so on. So, you can customize the way that your Thonny editor works this way.



Thonny options.

This also, if we go into Tools and Plugins, there’s a whole variety of plugins that you can install; some of them, as I said earlier, in version 3.3.4, comes built into Thonny itself. The ESP tool package allows the Thonny IDE to interact with the ESP32 and, for example, flash new firmware on it.

But there are others you can search on PyPI, which is the Python repository for packages, and see what else is available. I'm going to show you how to use that later. There's also a package manager like this, which also allows you to search in PyPI for Python packages that contain libraries or code that is shareable and that you can use. Again, I'm going to show you a little later how to install a PyPI package.

Get Thonny

To get Thonny, go to the Thonny website. You can have a quick look at this to get a rundown of the most important features.

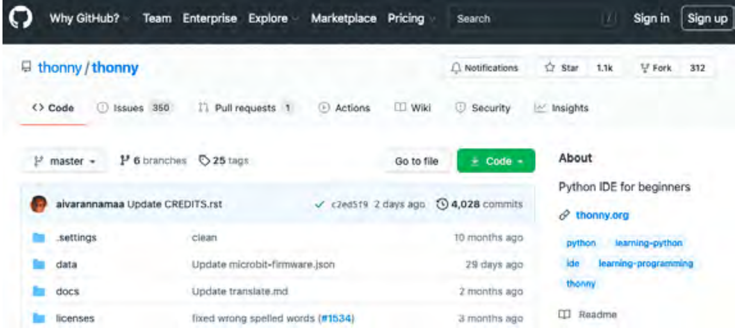
I highly recommend that you watch this video here. It's a demonstration of some of Thonny's most exciting features, especially the debugging features produced by one of the Thonny developers. So, do check it out.

To download Thonny, click on your operating system – in my case, I'm working on a Mac – download the file, double click on it, and install it. There's nothing special about it. It's straightforward.

Download Thonny from thonny.org.

Thonny Github repository

Another web resource I want to show you is the GitHub repository. So, you can see, as I said, the source code of the Thonny project. Now, here you will find additional releases. So, click on the Releases link, and it will take you to a page where you can access not just the latest release, 3.3.4, in my case. 3.3.5 is just being worked on at the moment. It's not available via the download button here. You can see this is still 3.3.4. The bleeding edge version is 3.3.5.



The Thonny GitHub repository.

But I found on the Mac in particular; if you are using macOS Big Sur, which is macOS 11, then version 3.3.4 does not work correctly. You may need to go to an older version; let's say, 3.3.3 did work for me. So, in case you need an older version, this is where you can get it from.

Learn MicroPython for the ESP32

With this video course, you will learn how to use theMicroPython programming language with the ESP32 micro-controller.

MicroPython is the perfect language for anyone looking for the easiest (yet still powerful) way to program a micro-controller.

__CONFIG_colors_palette__{“active_palette”:0,”config”:{“colors”:{“3e1f8”:{“name”:”Main Accent”,”parent”:-1}},”gradients”:[]},”palettes”:[{“name”:”Default Palette”,”value”:{“colors”:{“3e1f8”:{“val”:”rgb(217, 49, 33)”}},”gradients”:[]},”original”:{“colors”:{“3e1f8”:{“val”:”rgb(19, 114, 211)”,”hsl”:{“h”:210,”s”:0.83,”l”:0.45}}},”gradients”:[]}}]}__CONFIG_colors_palette__

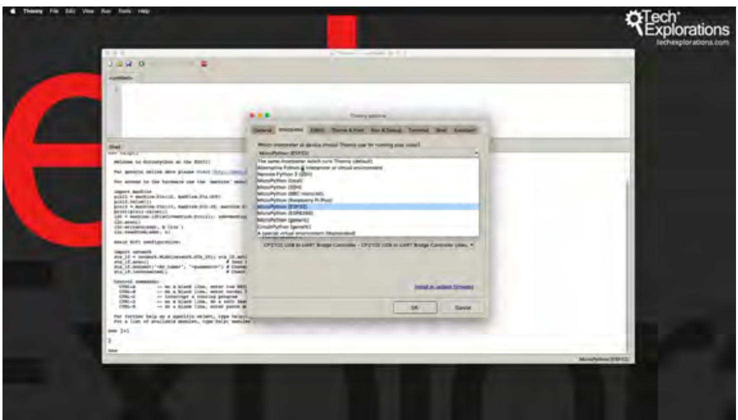
---

7. Set the Python interpreter

MICROPYTHON WITH THE ESP32 GUIDE SERIES

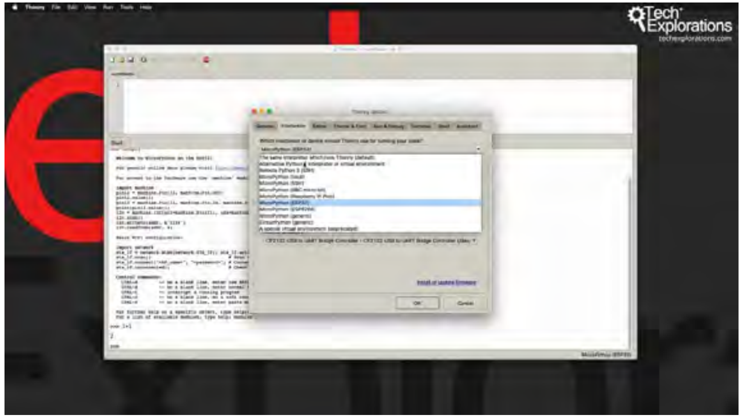
How To Set The MicroPython Interpreter

Thonny IDE can work with multiple Python interpreters. Not at the same time, of course, but Thonny does give you the ability to select which interpreter you want to use next, also which device that interpreter is installed on.



In this lesson I will show you how you can switch between interpreters quickly.

The interpreter selector



The interpreter selector is in the Thonny options window.

Let’s start.

Go to Tools and then Options, you’ll see that under the interpreter tab, expand that drop-down menu, and you’ll see that Thonny comes equipped with a variety of interpreters. There’s an interpreter that ships with Thonny itself. It runs as part of the Thonny environment, but you can also choose to use the Python instance installed on your computer.

Of course, you can run MicroPython on various devices, including BBC micro:bit, Raspberry Pi Pico, and the ESP32.

I have already selected the ESP32 since we installed the MicroPython firmware on my brand new device, and we tested it.

```
  CTRL-B        -- on a blank line, enter normal REPL mode
  CTRL-C        -- interrupt a running program
  CTRL-D        -- on a blank line, do a soft reset of the board
  CTRL-E        -- on a blank line, enter paste mode

For further help on a specific object, type help(obj)
For a list of available modules, type help('modules')

>>> 1+1

2

>>>

MicroPython v1.14 on 2021-02-02; ESP32 module with ESP32
Type "help()" for more information.
>>>
```

The MicroPython prompt confirms your selected interpreter.

So, we’ve got the MicroPython prompt; this is information about the Python interpreter we are using. So, if I do a simple calculation in Python, like:

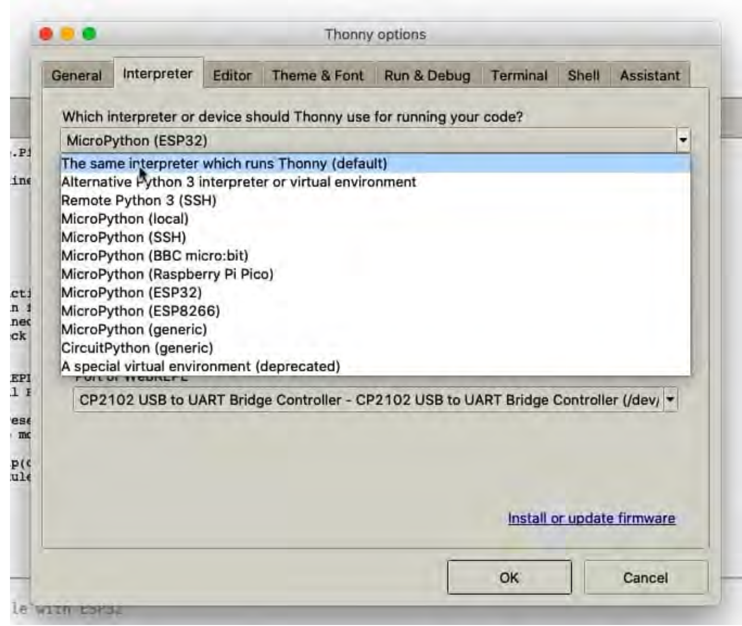
>>> 1 + 12>>>

… you will see that MicroPython on the ESP32 is working, and communicating with Thonny IDE.

Switch to a different interpreter

From here, I want to switch to the Thonny built-in environment.

You can do that via the Thonny options; click the Interpeter tab, and select “The same interpreter which runs Thonny” option from the list.



Switching to the Thonny IDE build-in Python interpreter.

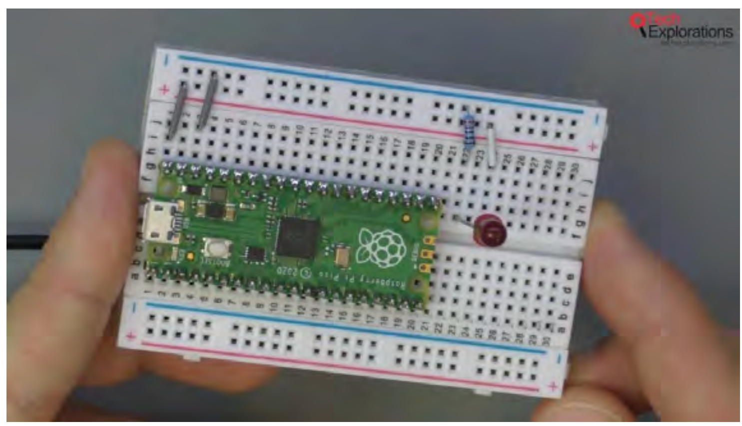
You’ll get a new prompt where you can type in the same simple calculation as before. The result, of course, is the same.

Just switched to default Python.

As you can see, the name of this environment they're working on right now is Python 3.7.9, which is the default Python for Thonny IDE, not MicroPython.

MicroPython for the Raspbery Pi Pico

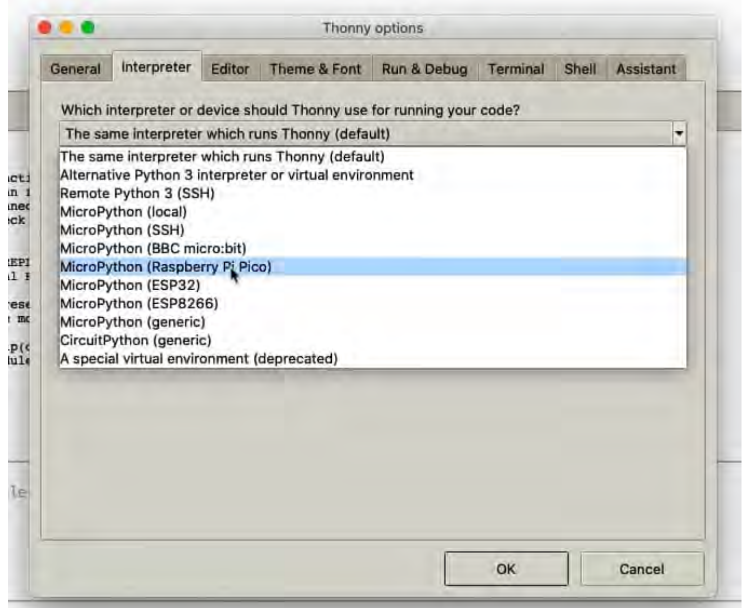
But how about something else? How about we try MicroPython on the new Raspberry Pi Pico?



The Raspberry Pi Pico

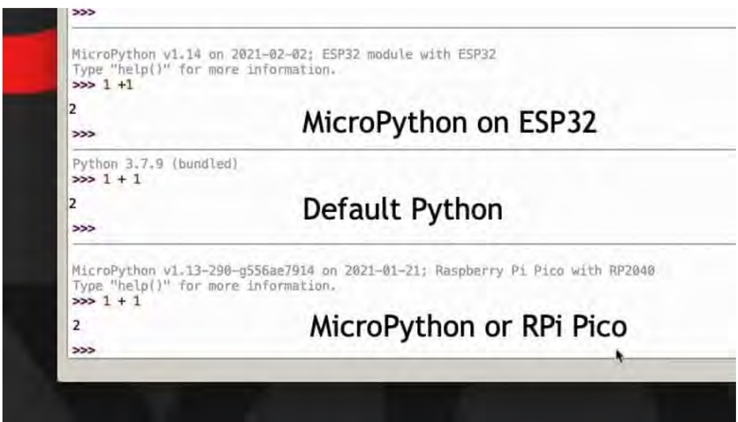
In the photo above you can see a Raspberry Pi Pico microcontroller that runs MicroPython. On the breadboard I have added an LED here, which is just showing me when power is connected.

Let’s see if we can program this board with MicroPython on Thonny IDE.



Thonny IDE supports the Raspberry Pi Pico "out of the box".

We'll go to Tools, Options, select the Raspberry Pi Pico. There's support for the Pico.



MicroPython on Raspberry Pi Pico.

After selecting the interpreter for MicroPython on a Raspberry Pi Pico, the REPL prompt becomes available so we can start our interaction with it.

I've have created several lectures in this course where I demonstrate how to use MicroPython on the Raspberry Pi Pico in depth. I didd the same with the BBC Micro:bit.

Learn MicroPython for the ESP32

With this video course, you will learn how to use theMicroPython programming language with the ESP32 micro-controller.

MicroPython is the perfect language for anyone looking for the easiest (yet still powerful) way to program a micro-controller.

__CONFIG_colors_palette__{“active_palette”:0,”config”:{“colors”:{“3e1f8”:{“name”:”Main Accent”,”parent”:-1}},”gradients”:[]},”palettes”:[{“name”:”Default Palette”,”value”:{“colors”:{“3e1f8”:{“val”:”rgb(217, 49, 33)”}},”gradients”:[]},”original”:{“colors”:{“3e1f8”:{“val”:”rgb(19, 114, 211)”,”hsl”:{“h”:210,”s”:0.83,”l”:0.45}}},”gradients”:[]}}]}__CONFIG_colors_palette__

Install MicroPython On The ESP32

A new ESP32 board does not (normally) come with the MicroPython firmware installed. Before you can use it, you'll need to install the MicroPython firmware. In this lesson I'll show you how to do it.

At this point, you should already have installed Thorney IDE on your computer. If you haven't done so, you should do it now.

A brand new ESP32 does not normally come with the MicroPython firmware pre-installed. Before you can actually

start working with MicroPython on this microcontroller is to install the MicroPython firmware.

Installing the MicroPython firmware

To install the MicroPython firmware to an ESP32, you need two things:

First, you need to download the firmware for the particular device from the Micro Python website.

And second, is to use a appropriate tool to upload the firmware binary file to your ESP32.

Luckily, the newer versions of the Thonny IDE come equiped with the upload tool.

Download the MicroPython firmware

To download the firmware for the ESP32, go to download page on the MicroPython website. Beware, there is a specific firmware file for each supported target board. You can’t upload a Micro:bit firmware to an ESP32.

MicroPython FORUM DOCS QUICK-REF DOWNLOAD STORE CONTACT

Firmware for Generic ESP32 module

The following files are daily firmware for ESP32-based boards, with separate firmware for boards with and without external SPIRAM. Non-SPIRAM firmware will work on any board, whereas SPIRAM enabled firmware will only work on boards with 4MiB of external pSRAM.

Program your board using the esptool.py program, found here. If you are putting MicroPython on your board for the first time then you should first erase the entire flash using:

```
esptool.py --chip esp32 --port /dev/ttyUSB0 erase_flash
```

From then on program the firmware starting at address 0x1000:

```
esptool.py --chip esp32 --port /dev/ttyUSB0 --baud 460800 write_flash -z 0x1000
```

Download the MicroPython firmware for the ESP32 from micropython.org

Once at the download page, find the list with the ESP-IDF v4.x downloads, and select the latest stable firmware. At the time I am writing this, the latest version is 1.14.

Firmware with ESP-IDF v4.x

Firmware built with ESP-IDF v4.x, with support for BLE and PPP, but no LAN.

- GENERIC : esp32-20210414-unstable-v1.14-161-g2ac09c269.bin
- GENERIC : esp32-20210413-unstable-v1.14-159-g1a2ffda17.bin
- GENERIC : esp32-20210412-unstable-v1.14-157-g2668337f3.bin
- GENERIC : esp32-20210412-unstable-v1.14-153-g22554cf8e.bin
- GENERIC : esp32-idf4-20210206-unstable-v1.14-9-g9dedcf122.bin
- GENERIC : esp32-idf4-20210205-unstable-v1.14-8-g1f800cac3.bin
- GENERIC : esp32-idf4-20210204-unstable-v1.14-3-g7c4435459.bin
- GENERIC : esp32-idf4-20210204-unstable-v1.14-1-g7f7b4f2bc.bin
- GENERIC : esp32-idf4-20210202-v1.14.bin
- GENERIC : esp32-idf4-20200902-v1.13.bin
- GENERIC : esp32-idf4-20191220-v1.12.bin
- GENERIC-SPIRAM : esp32spiram-20210414-unstable-v1.14-161-g2ac09c269.bin
- GENERIC-SPIRAM : esp32spiram-20210413-unstable-v1.14-159-g1a2ffda17.bin

Opt for a stable firmware, unless you know what you are doing.

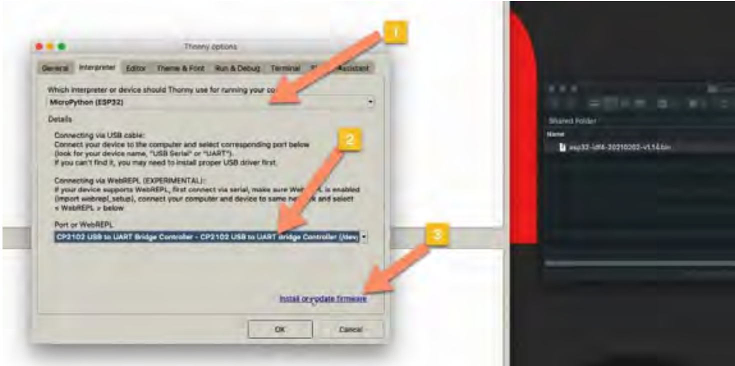
There are unstable versions and there are also versions for ESP32 with the additional SPI RAM chip, which provides additional memory. I don’t have that, so I’m going to go with the latest generic version.

Download the file, and continue with Thonny.

Flash the MicroPython firmware to the ESP32

Connect your ESP32 to your computer, then in Thonny select Tools, Options to bring up the Options dialog box.

Click on the Interpreter tab. There are two drop-down menus here.

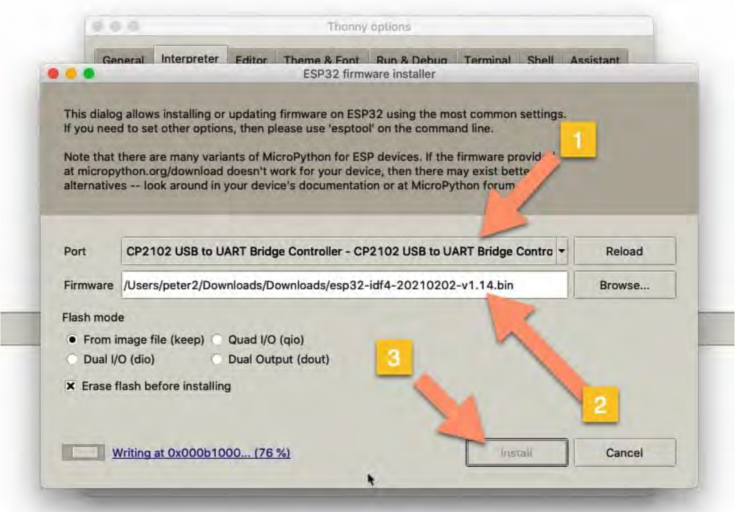


The firmware upload tool in Thonny IDE.

The top one is the interpreter selector. Select the “MicroPython (ESP32)” from the list.

The second is the port selector. Select the port to which your ESP32 is connected.

Finally, click on the “install or update” hyperlink.



Select the firmware file to flash to your ESP32.

This will take you to the ESP32 firmware installer box. Again, you will need to select the appropriate communications port.

Then, browse for the firmware file that you downloaded earlier.

There are a few options for the installer, which I recommend that you leave in their default states (see my screenshot above).

Finally, click on the Install button to start the flashing process. The process takes around one minute to complete.

Test your new MicroPython interpreter

Once the flashing is complete, your ESP32 MicroPython interpreter will connect to Thonny, and the REPL prompt will appear.

The REPL prompt from the brand-new MicroPython interpreter on the ESP32.

Type “help()” to see helpful information from the interpreter.

And with this, your ESP32 board can now work with MicroPython.

Learn MicroPython for the ESP32

With this video course, you will learn how to use theMicroPython programming language with the ESP32 micro-controller.

MicroPython is the perfect language for anyone looking for the easiest (yet still powerful) way to program a micro-controller.

__CONFIG_colors_palette__{“active_palette”:0,”config”:{“colors”:{“3e1f8”:{“name”:”Main Accent”,”parent”:-1}},”gradients”:[]},”palettes”:[{“name”:”Default Palette”,”value”:{“colors”:{“3e1f8”:{“val”:”rgb(217, 49, 33)”}},”gradients”:[]},”original”:{“colors”:{“3e1f8”:{“val”:”rgb(19, 114, 211)”,”hsl”:{“h”:210,”s”:0.83,”l”:0.45}}},”gradients”:[]}}]}__CONFIG_colors_palette__

8. How to write and execute a MicroPython program

MicroPython with the ESP32 guide series

How to write and execute a MicroPython program

In this lesson, I will show you how to write and execute a simple MicroPython program. I'll do this with Thonny IDE in two different ways.

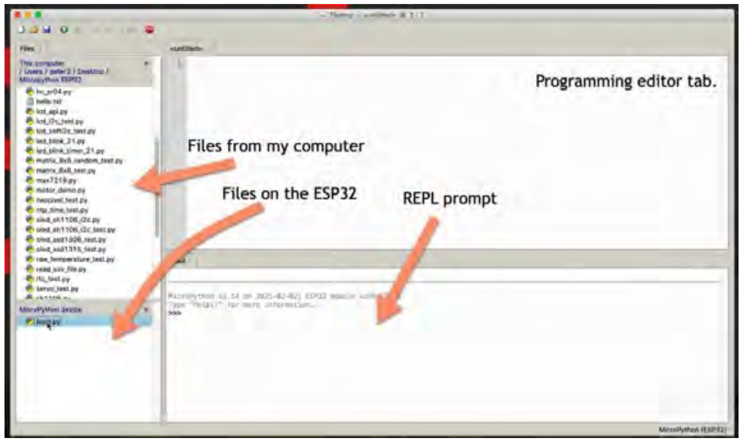
I'll show you how to run a simple MicroPython program that prints "Hello World from MicroPython." in the shell.

First, I'll show you how to write the program in the REPL and in a new file tab.

Next, I'll show you how to load an existing file that contains a MicroPython program from your computer's file system.

Run a program on the REPL

First, start your Thonny IDE editor. Open the file browser tools in the left side of the editor. Ensure that your ESP32 is connected to your computer. Files stored on the ESP32 will appear in the lower left side of the editor.



The main components of the Thonny editor.

If you have just flashed your ESP32 with the MicroPython interpreter, you will only see the boot.py file in the MicroPython device segment.

Let’s write a very simple program.

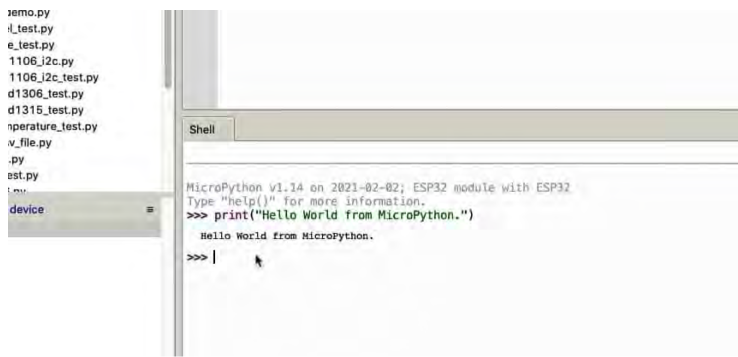
First, we’ll run the program on the Shell. Then, we’ll in which we’ll store the same program so that we can run it in the future without having to write it from scratch.

Here’s the program:

```
>>> print(“Hello World from MicroPython.”)

Hello World from MicroPython.
```

Simple, right?

When you type this program in the REPL and hit the Enter key, the interpreter will execute it and show the response in the shell.

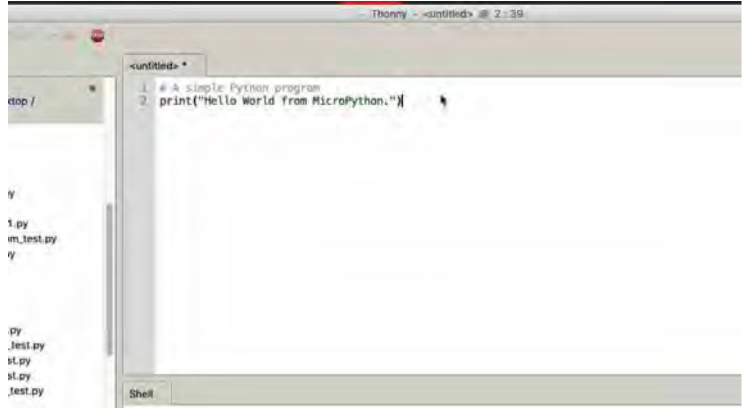


The interpreter has just executed my one-line program.

Remember, this single-line program was executed on the ESP32, not my host computer. When you type a command in the REPL and hit the Enter key, the command is evaluated by the REPL running on the target device immediately.

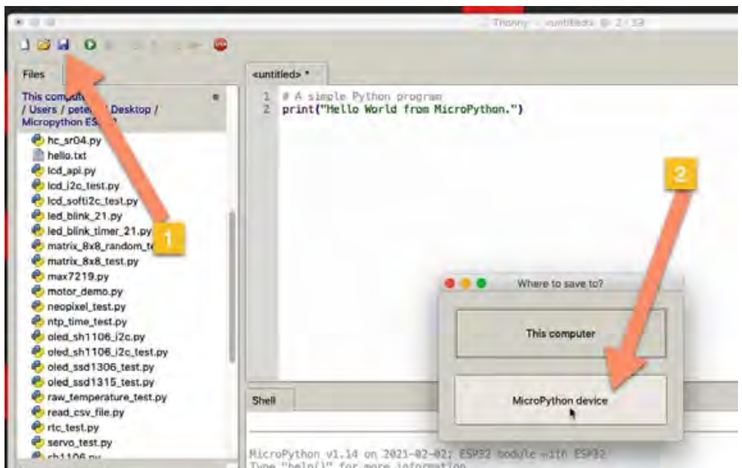
Run a program from a file

Another way to execute MicroPython programs, particularly useful for programs that are more than a few lines in size, is, of course, to store them in a file. So, I just copied my single command, my very small program, into a file. The program contains two lines. The first one is a commend, and starts with the “#” symbol. The second line is the actual instruction that prints out some text.



A tiny MicroPython program.

To save the program, click on the icon that looks like an old-fashioned floppy disk.



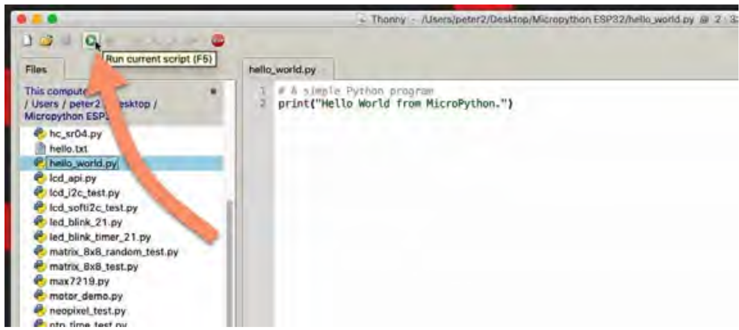
To save a program, click on the floppy disk button and select the target.

Thonny gives me a choice of where is it that I'd like to save this program, either my computer or the MicroPython device. In this instance, I'm going to go for the computer and that will

give me the option to store it somewhere. So, let's say I'm going to put it on this location and just say helloworld.py as the file system and save that.

Note: regardless of where you save the file, execution will always take place on the target device.

To run the program from the file, click on the green "play" button.



Press "play" to run a program.

There are few nuisances in this system that we are going to explore a little later. Those nuisances have to do with dependencies.

For example, what if there is a module that is required by this program which is not stored on the MicroPython device? Then, you're not going to just be able to upload and execute this file on the device. You will also need to take care of those dependencies.

And there are a few examples later on in this course where I show you how that works.

All right. Obviously, we're going to do a lot more into how to write and execute programs on the ESP32 using MicroPython. But in this quick introduction, I just want to show you the simplest possible way of doing that.

Learn MicroPython for the ESP32

With this video course, you will learn how to use theMicroPython programming language with the ESP32 micro-controller.

MicroPython is the perfect language for anyone looking for the easiest (yet still powerful) way to program a micro-controller.

__CONFIG_colors_palette__{“active_palette”:0,”config”:{“colors”:{“3e1f8”:{“name”:”Main Accent”,”parent”:-1}},”gradients”:[]},”palettes”:[{“name”:”Default Palette”,”value”:{“colors”:{“3e1f8”:{“val”:”rgb(217, 49, 33)”}},”gradients”:[]},”original”:{“colors”:{“3e1f8”:{“val”:”rgb(19, 114, 211)”,”hsl”:{“h”:210,”s”:0.83,”l”:0.45}}},”gradients”:[]}}]}__CONFIG_colors_palette__

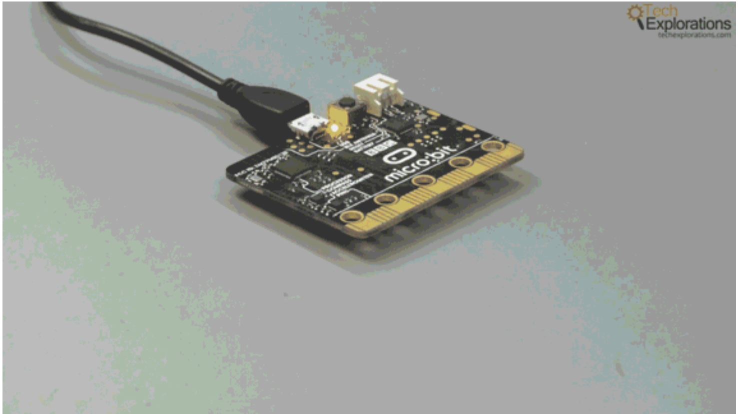
---

10. Thonny IDE with BBC micro:bit

MICROPYTHON WITH THE ESP32 GUIDE SERIES

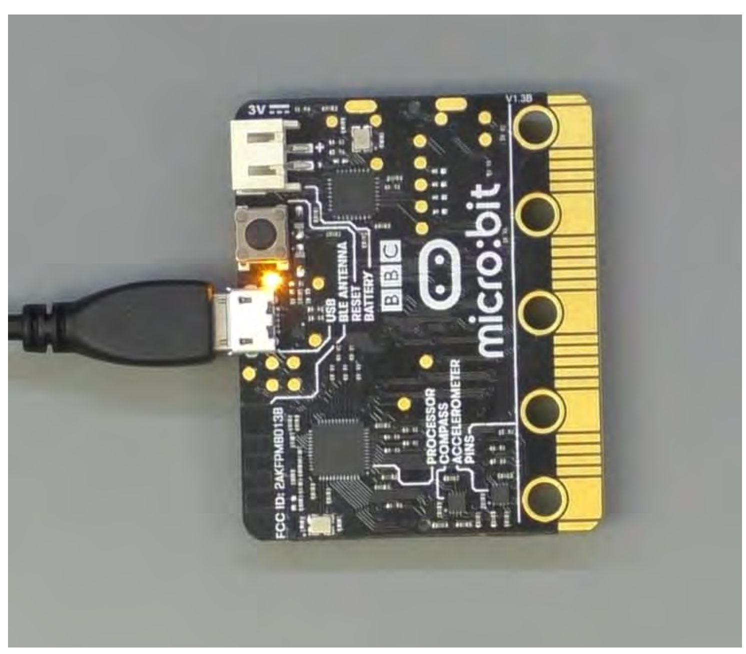
Thonny IDE With BBC Micro:bit

In this lesson, I'll show you how to run a simple MicroPython program on the BBC micro:bit that scrolls text on the device's 5 by 5 bitmap display.

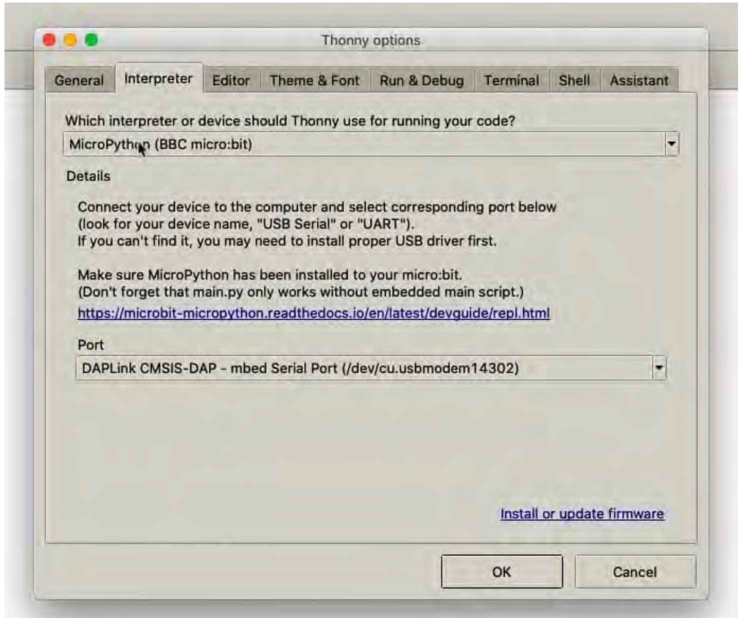


Of course, we'll do that using MicroPython and Thonny IDE.In this lesson, we'll take a closer look at the boards that can use MicroPython.

Setup



The BBC Micro:bit, connected via USB.



Thonny IDE has build-in support for the Micro:bit.

The Micro:bit is equiped with a 5×5 LED matrix display on its rear side. Therefore, there is no wiring to be done. Simply connect the device to your computer via a USB cable.

Next, start Thonny and navigate to the Options dialog box. Ensure that MicroPython and BBC micro is selected as the interpreter for this session. Then, select the appropriate port.

Install the MicroPython interpreter on the Micro:bit

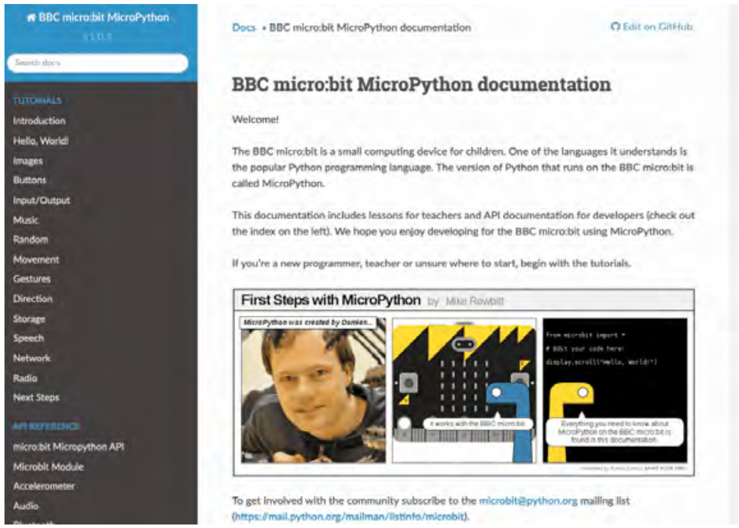
The micro:bit does not come from factory with the MicroPython interpreter installed on it.

So, if you are not able to make this work with your Thonny IDE, once you have selected the interpreter in the port, click on the

install or update firmware in order to go ahead and install the MicroPython interpreter on the micro:bit.

I've already done that, so I'm not going to overwrite my firmware so I will click the cancel button. But in your case, you may need to do that if this is the first time that you are connecting your micro:bit to your computer and wanting to use it as a MicroPython interpreter in the target device. Follow the same process I have documented in the ESP32 flashing lesson.

Micro:bit documentation for MicroPython



The documentation for MicroPython on the BBC Micro:bit.

Another resource that is very useful and I encourage you to look at if you are interested in using the BBC micro:bit as a MicroPython device is to look at the BBC micro:bit MicroPython documentation.

There's a lot that you can do with the micro:bit. The micro:bit does come with a lot of onboard hardware, such as an

accelerometer.

It comes with two programmable buttons, a 5×5 matrix led display. It also has a row of multi-purpose input/output pins.

And the documentation shows you how to use all of that hardware.

Hello World on the Micro:bit with MicroPython

The 5×5 LED matrix display on the Micro:bit

In this example, I'll show you how to write a simple program that prints out "Hello World" in a way that it takes the individual letters scroll across the screen.

The MicroPython instruction for this purpose is microbit.display.scroll. The “dot” notation format of this instruction indicates that the scroll function, is inside the display module, which itself is inside the microbit package.

Below you can see an extract from the documentation, showing the scroll function and its parameters.

microbit.display.scroll(*value, delay=150, *, wait=True, loop=False, monospace=False*)

Scrolls `value` horizontally on the display. If `value` is an integer or float it is first converted to a string using `str()`. The `delay` parameter controls how fast the text is scrolling.

If `wait` is `True`, this function will block until the animation is finished, otherwise the animation will happen in the background.

If `loop` is `True`, the animation will repeat forever.

If `monospace` is `True`, the characters will all take up 5 pixel-columns in width, otherwise there will be exactly 1 blank pixel-column between each character as they scroll.

Note that the `wait`, `loop` and `monospace` arguments must be specified using their keyword.

Documentation extract for the “scroll” function.

As per the documentation, the only required value is a string in the first parameter. The rest are optional and they have their own default value.

Here’s my program:

>>> import microbit>>> microbit.display.scroll(“Hello World!”)

Go ahead and type this program in the Thonny editor shell (no need to create a file for such a small program).

This is what the program looks like in Thonny:

The “Hello World!” program in Thonny.

Because we typed the program in the shell, the MicroPython interpreter will execute the scroll function as soon as you hit the return key.

Turn over the Micro:bit so that you can see the LED matrix display, and hit the return key. You should see the text scrolling across the screen, like this:

Text scrolling on the 5×5 matrix display of the Micro:bit

Controlling individual pixels

Of course, it is possible to control each pixel on the LED matrix display individually. To do this, you can use the “set_pixel” function.

Here’s the extract from the documentation that provide information about this function:

microbit.display.set_pixel(*x*, *y*, *value*)

Set the brightness of the LED at column `x` and row `y` to `value`, which has to be an integer between 0 and 9.

Documentation extract for the “set_pixel” function.

The first two parameters are the x and y coordinates of a pixel, and the third paramter is the intensity (“0” will turn off the LED, “9” will turn it on at maximum intensity).

Here’s a small program that turns on two of the pixels of the display:

>>> import microbit>>> microbit.display.set_pixel(0,0,9)>>> microbit.display.set_pixel(4,4,9)>>> microbit.display.clear()

As you type the two “set_pixel” command into the shell, and hit the return key, the two pixels at the top-left and bottom-right of the display will turn on. The display should look like this:

Two LEDs are turned on.

To clear the display and turn off all LEDs, simply call the "clear" function.

This was a quick demonstration of how you can use the display on the BBC micro:bit with a simple MicroPython script.

Learn MicroPython for the ESP32

With this video course, you will learn how to use theMicroPython programming language with the ESP32 micro-controller.

MicroPython is the perfect language for anyone looking for the easiest (yet still powerful) way to program a micro-controller.

__CONFIG_colors_palette__{"active_palette":0,"config":{"color

s”:{“3e1f8”:{“name”:”Main Accent”,”parent”:-1}},”gradients”:[]},”palettes”:[{“name”:”Default Palette”,”value”:{“colors”:{“3e1f8”:{“val”:”rgb(217, 49, 33)”}},”gradients”:[]},”original”:{“colors”:{“3e1f8”:{“val”:”rgb(19, 114, 211)”,”hsl”:{“h”:210,”s”:0.83,”l”:0.45}}},”gradients”:[]}}]}__CONFIG_colors_palette__

Thonny IDE Advanced configuration

In this lesson, I'd like to show you the advanced configuration file of Thonny IDE so that you may modify some of the functionality and that is not possible to do via graphical user interface.

Thonny has an advanced configuration file titled "configuration.ini", found in the Thonny application folder. In this file, you can change settings that control things such as how code is send to the target board, the size of a block, and whether to update the target device RTC (reat time clock) every time Thonny connects to the device.

Documentation

Advanced configuration

There are some hidden configuration options, which most of the users don't need to tweak, but which may be useful in some cases. In order to change them, you need to find the location of *configuration.ini* (Tools => Open Thonny data folder), close Thonny and edit the file by hand.

Here is an example section for ESP32 back-end:

```
[ESP32]
submit_mode = raw
write_block_size = 255
write_block_delay = 0.03
sync_time = True
utc_clock = False
```

- `submit_mode` controls how code is sent to the board. Thonny 3.2 sent it via `raw` mode, Thonny 3.3.0 and 3.3.1 via `paste` mode and since 3.3.2 the default is `raw_paste` if the device supports it (a new mode appearing in MicroPython 1.14) or `raw` as fallback. If you want `paste` mode as fallback, then you can specify it here. (Note, that some devices have problems with `paste` mode, see https://github.com/thonny/thonny/issues/1461 for an example.)
- `write_block_size` and `write_block_delay` control how the code is submitted in raw mode and for submitting data to stdin (`write_block_size` is also used with `paste` mode). Thonny will break the data into blocks of `write_block_size`

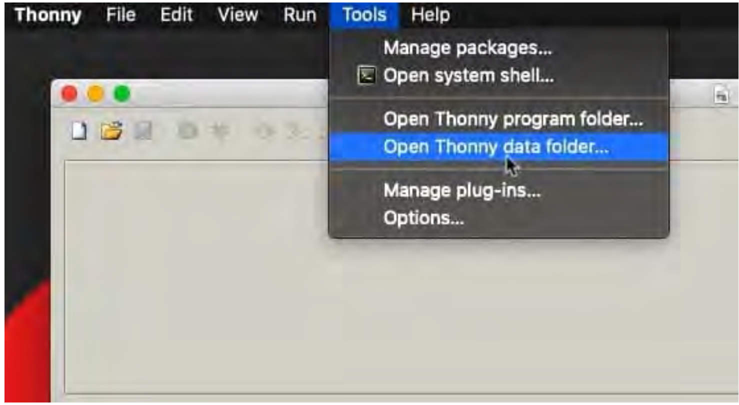
Thonny advanced configuration options are documented in the

project Wiki on GitHub.

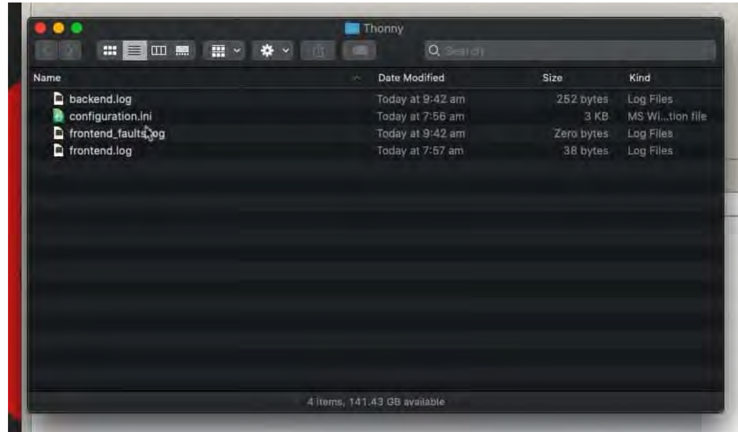
To begin with, go to the Thonny project on GitHub.

Then, scroll down to find the link for the wiki and click on it. Look for “MicroPython” in the table of contents and click on it. Once you reach the MicroPython page, scroll down to the advanced configuration section. This provides information about the advanced configuration options available for the ESP32.

Thonny data folder



A shortcut to the data folder is in the Tools menu.



The Thonny data folder.

The easiest way to find the advanced configurations file is to use the Thonny shortcut under the Tools menu. Select “Tools”, “Open Thonny data folder…”.

This will bring up the folder that contains the “configuration.ini” file.

Open this file with a text editor. It looks like this:

```
[general]
configuration_creation_timestamp = 2021-02-04T13:04:42.947341
language = en_US
environment = []
single_instance = True
event_logging = False
disable_notification_sound = False
debug_mode = False
ui_mode = regular
scaling = default
font_scaling_mode = default

[view]
full_screen = False
maximize_view = False
shellview.visible = True
filesview.visible = True
files_split = 536
assistantview.visible = False
astview.visible = False
```

The configuration.ini file with default settings.

Interesting configuration settings

The settings that I'm more interested at the moment are the sync time and the UTC clock time.

As you may know, the ESP32 has a real time clock integrated into the chip. When you use Thonny to upload a program, it is possible for Thonny to reset the clock to the correct system time and date.

To make that work, you make sure that the sync time keyword is set to true.

Another thing that you can consider doing, whether you want the real time clock to be set to UTC time or to your local system time, then you can control that via the UTC clock keyword. When you say false, then the RTC of your ESP32 will be synchronised to your computer's local time.

In Section 12 of the MicroPython course, I have a couple of lectures where I show you how to set the time in the RTC of your ESP32 programmatically, both manually and by getting accurate time and date from an internet atomic clock.

Learn MicroPython for the ESP32

With this video course, you will learn how to use theMicroPython programming language with the ESP32 micro-controller.

MicroPython is the perfect language for anyone looking for the easiest (yet still powerful) way to program a micro-controller.

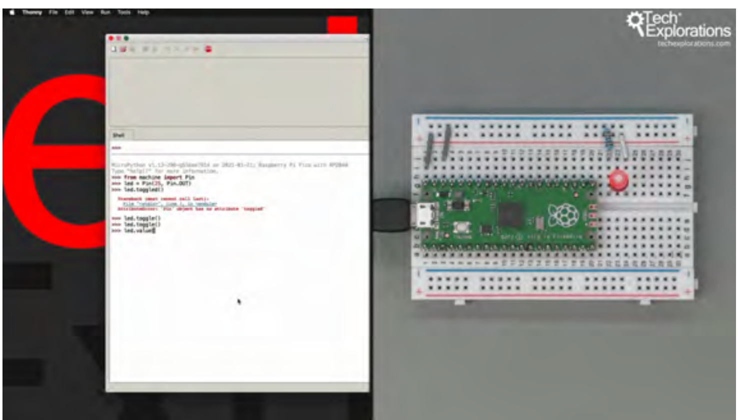
__CONFIG_colors_palette__{“active_palette”:0,”config”:{“colors”:{“3e1f8”:{“name”:”Main Accent”,”parent”:-1}},”gradients”:[]},”palettes”:[{“name”:”Default Palette”,”value”:{“colors”:{“3e1f8”:{“val”:”rgb(217, 49, 33)”}},”gradients”:[]},”original”:{“colors”:{“3e1f8”:{“val”:”rgb(19, 114, 211)”,”hsl”:{“h”:210,”s”:0.83,”l”:0.45}}},”gradients”:[]}}]}__CONFIG_colors_palette__

9. Thonny IDE with Raspberry Pi Pico

MICROPYTHON WITH THE ESP32 GUIDE SERIES

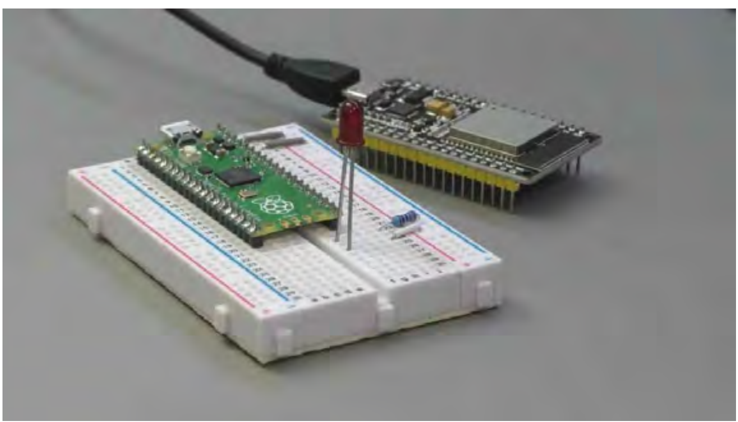
Thonny IDE With The Raspberry Pi Pico

In this lesson, I will demonstrate how to use Thonny IDE and MicroPython on a Raspberry Pi Pico.



In this lessomn, I will demonstrate how to use Thonny IDE and MicroPython on a Raspberry Pi Pico to do something simple with the Raspberry Pi Pico, which is, in this case, to make the onboard LED blink.

Setup the experiment



Let’s begin be connecting the Raspberry Pi Pico to the computer via the USB cable. Ensure that Thonny is running. Thonny should detect the Raspberry Pi Pico.

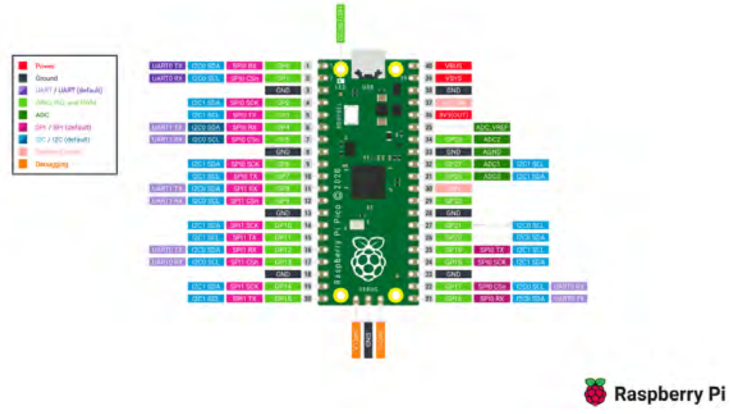
In the Thonny Tools menu, select “options”, then “interpreter”, and change the interpreter to the “MicroPython (Raspberry Pi

Pico)” option. Also select the correct connection port for the device.

The Raspberry Pi Pico comes with MicroPython already flashed. This means that it is literally “plug and play”; your don’t have to install MicroPython like you did with the ESP32.

Documentation

You can find information about the Raspberry Pi Pico on its web page. Scroll down on the board specifications, where you’ll see the pin map out.



The Raspberry Pi Pico is well-equipped with all sorts of GPIO and communications capabilities.

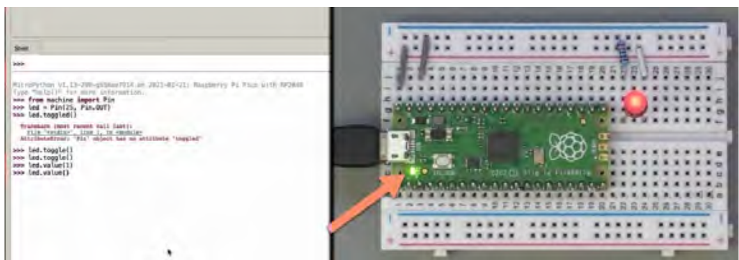
One interesting technology that comes with it is a programmable input/output or PIO state machines, which allow you to write programs and execute on specific GPIOs. Because they run directly on the co-processors of the GPIO, they’re not occupying any MCU cycles that are very, very fast. This is a fairly advanced topic, though, but I thought I should mention it because it’s really a feature that stands out when compared to other microcontroller units. If you are curious about PIO, see the RP2040 documentation (PDF) and browse to pave 338

(section 3.3. PIO Assembler).

Experiment

In this simple example, I will show you how to toggle the state of the built-in LED. Just as you can see in the pin map out, it is connected to GP25 using Thonny IDE.

Here what this looks in Thonny IDE, with the resulting lit onboard LED:



Here’s the code showing in the Thonny IDE shell:

```
>>> from machine import Pin>>> led = Pin(25, Pin.OUT)>>>
led.toggle()>>> led.toggle()>>> led.value(0)>>>
led.value(1)
```

First, import the Pin module from the machine library. The machine library is a library that is available for all microcontroller units that support MicroPython. It contains functions that are specifically created for the microcontroller that you’re targeting. In this case, the machine library contains functions that specifically apply to the Raspberry Pi Pico.So, now, that you have access to the Pin module, you can use the capabilities it provide to do things such as toggle the state of the LED

Next, create the LED object and set pin 25 as an output. are targeting GP25Now, that you have this object, you can use the toggle() function to turn it on or off.

You can also use the “value” function to explicitly control the state of a pin.

You can also check the state of the LED by calling the value, but without a parameter.

In the next lesson I’ll show you how to do something similar with the BBC Micro:bit. The purpose of these couple of lessons is to show you how versatile MicroPython. It makes it possible to jump from one kind of hardware to another with some relatively small modifications to your MicroPython program.

Learn MicroPython for the ESP32

With this video course, you will learn how to use theMicroPython programming language with the ESP32 micro-controller.

MicroPython is the perfect language for anyone looking for the easiest (yet still powerful) way to program a micro-controller.

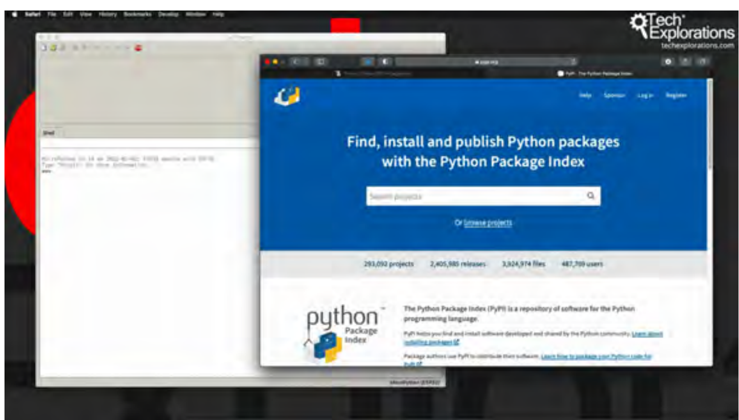
__CONFIG_colors_palette__{“active_palette”:0,”config”:{“colors”:{“3e1f8”:{“name”:”Main Accent”,”parent”:-1}},”gradients”:[]},”palettes”:[{“name”:”Default Palette”,”value”:{“colors”:{“3e1f8”:{“val”:”rgb(217, 49, 33)”}},”gradients”:[]},”original”:{“colors”:{“3e1f8”:{“val”:”rgb(19, 114, 211)”,”hsl”:{“h”:210,”s”:0.83,”l”:0.45}}},”gradients”:[]}}]}__CONFIG_colors_palette__

12. Find Python Packages at PyPI

MICROPYTHON WITH THE ESP32 GUIDE SERIES

Find Python Packages At PyPI

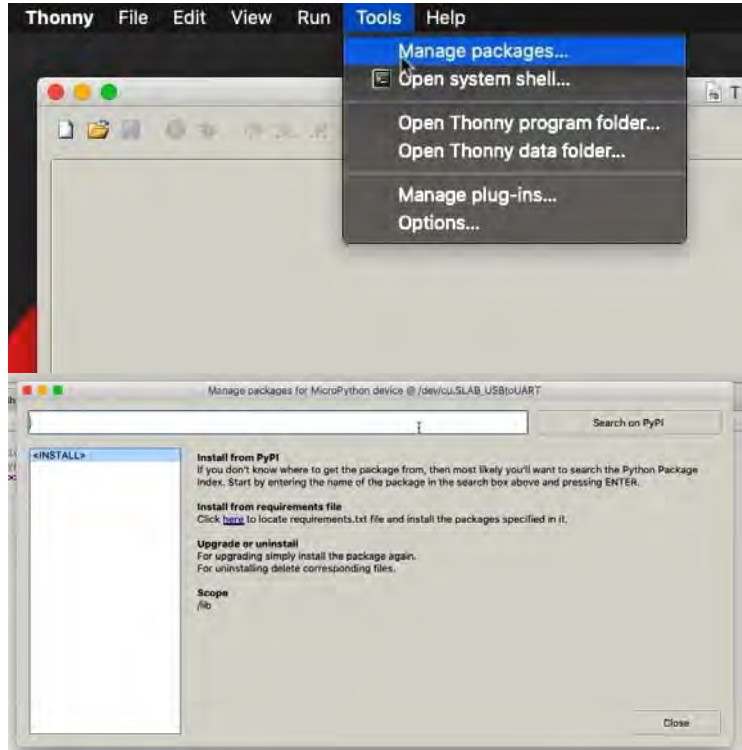
PyPi is a repository of Python and MicroPython packages. You can search PyPi from Thonny IDE and install packages with a single click. In this lesson I'll show you how.



The Thonny PyPi tool

You can install a Python package directly from the Thonny ID user interface.

You will find the PyPi search and installation tool under Tools, Manage Packages.



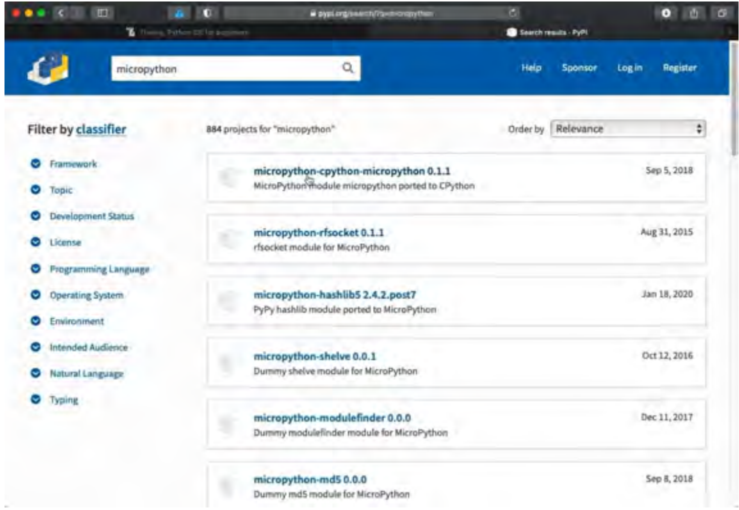
Example 1: search, find, install a MicroPython package

PyPi contains thousands of Python packages. Only some of them are specifically written in MicroPython. Most MicroPython packages are written to be compatible with a specific microcontroller, like the ESP32 or the Micro:bit.

You need to be mindful of this, and take care so that the package that you eventually download to your Thonny is actually compatible with your target device.

I'll show you an example of how to do this.

Let’s start with a very broad search on the term “MicroPython” directly on the PyPi website. This search will return any packages that contain the term “MicroPython”.



A generic search on PyPi for the term “MicroPython” returns 884 projects.

As you can see there are almost a thousand project that contain this name, I am not familiar with any of the projects showing in the first page of the list.

I’ll pick one randomly to take a closer look, from page two of the list.

The “lucky” project is titled “micropython-selectors 0.0.1“.

I found this package in PyPi. Looks intersting. Let's install it in Thonny.

Let's say that this is a package that you would like to use in your MicroPython project. The next thing to do is to download it and install it in Thonny. Copy the name, go to Thonny, bring up the Manage Packages window (under "Tools"), and copy the package name in the search field.

Search for a package in PyPi by name.

Click on the package name to see more informatio, then click on Install.

In the list of results, you'll find the package with the name you searched for at the top. Click on the package hyperlink to get to the package information page. Click on the Install button to download and install the package.

A few seconds later, you'll see the package listed in the left-side packages list.

The installed packages appear in the list on the left of the Manage Packages window.

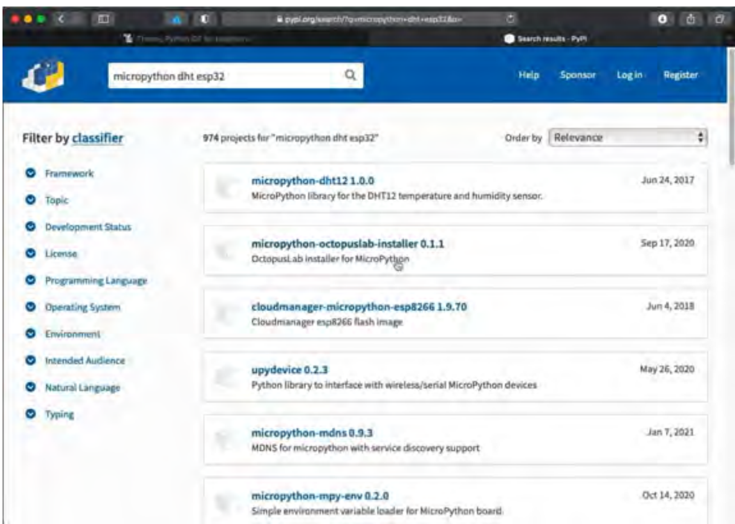
Example 2: search for a specific MicroPython package (gives error)

Let's look at another example. Unlike the previous generic search, now I'd like to be somewhat more specific. Let's look for a MicroPython package that is compatible with the ESP32.

Let's try this search term: "MicroPython dht ESP32".

With this search I hope to find a DHT11 or DHT22 MicroPython driver for the ESP32.

Do the search in PyPi.org and take a minute to look at the results.

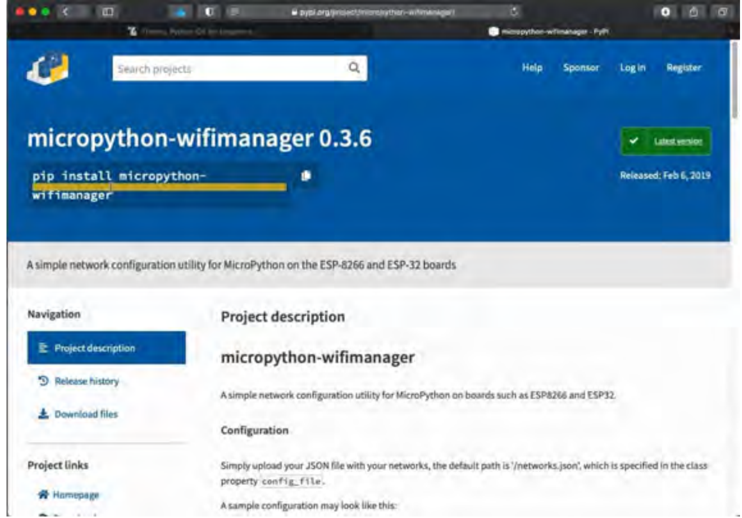


A search for "micropython dht esp32".

Because MicroPython can be used across a lot of different hardware targets, not all the package search results will be compatible with the target you intend to use. However, ESP23 and ESP8266 are (generally) cross-compatible when it comes to MicroPython. So, in general, if you find a package that is

marked to be compatible with the ESP8266, chances are that the package will also work with the ESP32.

In the result list for the term “micropython dht esp32”, there are several hits. One that drew my interest is titled “micropython-wifimanager“.

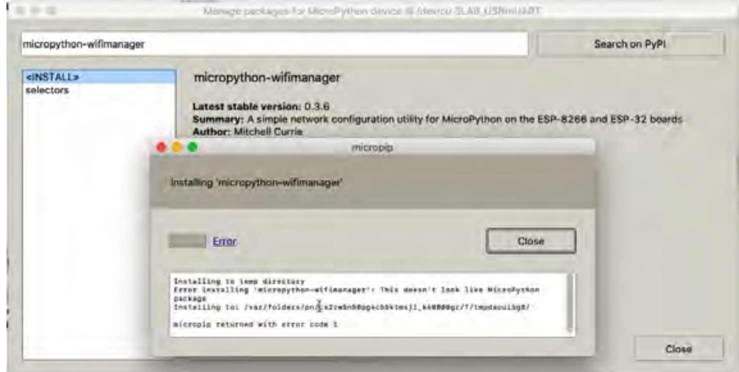


This package looks interesting, and is compatible with both ESP32 and ESP8266.

Let’s install this package in Thonny.

Follow the exact same process as with example 1. Copy the package name into the Thonny package manager tool, and click on the “Install” button.

Unfortunately, the installation failed for me at the time I was writing this guide (it may work for you).



This package cannot be installed in Thonny.

I am not sure why the installation failed, but the error message suggest a problem with the configuration of the package itself, and not Thonny.

This is unfortunate, but not the end of the world.

Let’s try something else.

Example 3: esp32-net-config

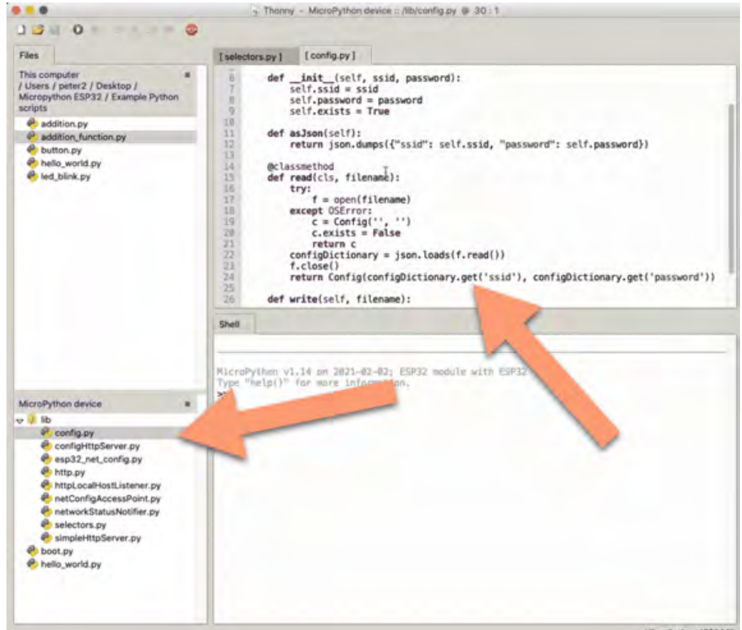
An interesting package is titled “esp32-net-config”. It creates a local WiFi access point so that you can configure a hot-spot SSID and password without having to hard-wire this information on your ESP32.

Install it as you did with examples 1 and 2. The package contains several files, that will be listed in the left text box:

The “esp32-net-config” package consists of several files.

Close the package manager, and look at the file browser of the ESP32 target device in the left bottom corner of the Thonny IDE. Inside the lib directory you can see the files that make up the package you just installed.

Click on the config.py file (or any of the other files, if you prefer), to look inside.



The contents of the config.py file, and the files that make up the “esp32-net-config” package listed in the ESP32 file browser.

If you want to curious to learn how to use this package, you can have a look at the information in the project’s GitHub repository which contains instructions and examples.These were three simple examples of how you can find, install, and inspect MicroPython packages available in the PyPi repository.

Learn MicroPython for the ESP32

With this video course, you will learn how to use theMicroPython programming language with the ESP32 micro-controller.

MicroPython is the perfect language for anyone looking for the easiest (yet still powerful) way to program a micro-controller.

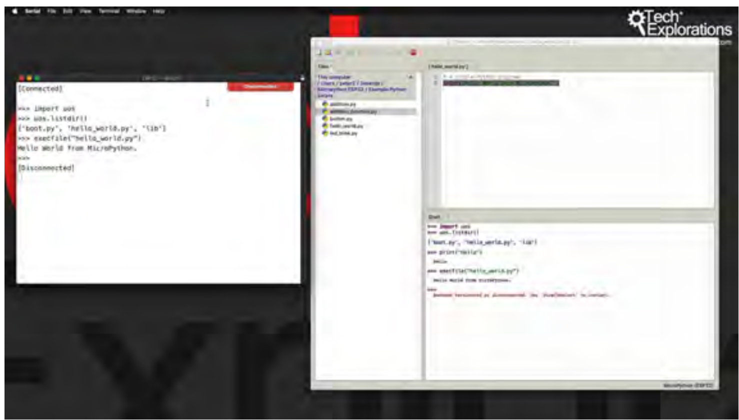
__CONFIG_colors_palette__{“active_palette”:0,”config”:{“colors”:{“3e1f8”:{“name”:”Main Accent”,”parent”:-1}},”gradients”:[]},”palettes”:[{“name”:”Default Palette”,”value”:{“colors”:{“3e1f8”:{“val”:”rgb(217, 49, 33)”}},”gradients”:[]},”original”:{“colors”:{“3e1f8”:{“val”:”rgb(19, 114, 211)”,”hsl”:{“h”:210,”s”:0.83,”l”:0.45}}},”gradients”:[]}}]}__CONFIG_colors_palette__

13. The MicroPython shell

MICROPYTHON WITH THE ESP32 GUIDE SERIES

The MicroPython Shell

In this lesson I'll show you a couple of ways to interact with the MicroPython Shell and run interactive programs or execute programs that are already stored on the ESP32 file system.



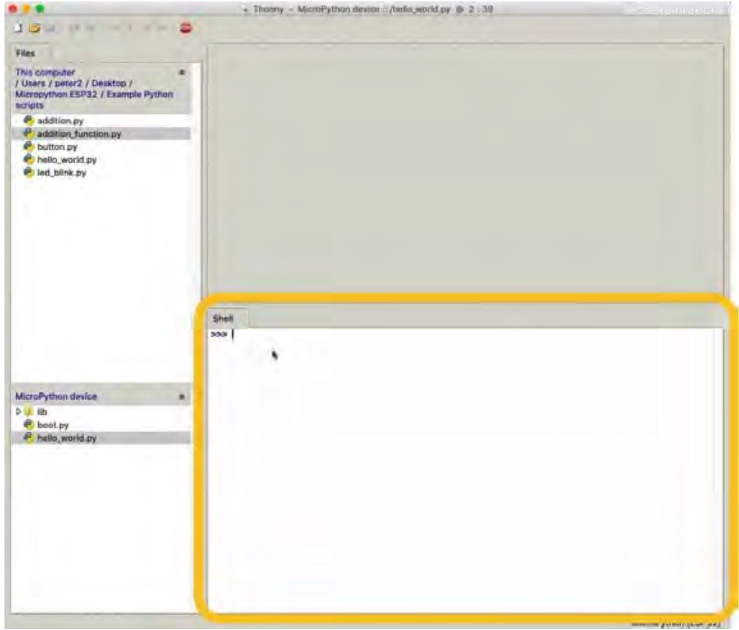
MicroPython provides an interactive shell that is running on the ESP32 (or any other compatible device, such as the Raspberry Pi Pico or the BBC Micro:bit).

You can connect to the MicroPython shell via a terminal tool like Putty or screen.

Of course, you can also use Thonny or even the Arduino IDE serial monitor.

In this lesson I will show you how to use Thonny and Serial. Both of these tools make it easy to interact with MicroPython via the shell.

Shell with Thonny



The Thonny IDE shell gives you access to the MicroPython interpreter that is running on the ESP32.

Connect your ESP32 to your computer and start Thonny. Ensure that the correct port is selected (see lesson 5).

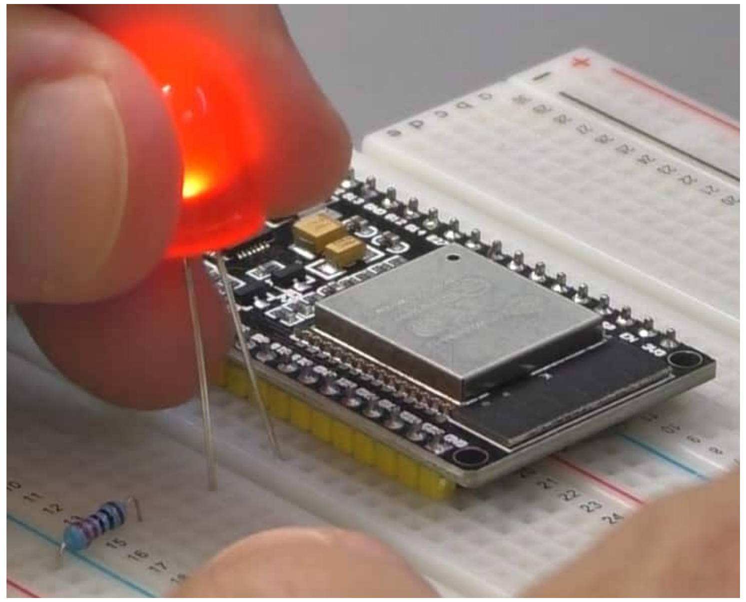
Once your ESP32 is connected, you will see the MicroPython shell prompt in the "Shell" tab, located at the lower half of the Thonny editor. The prompt looks like this (three greater-then symbols):

```
>>>
```

The Shell, is running on the ESP32, not Thonny.

Thonny simply provides a tool to access the Shell.The shell is now active and waiting for me to type in an instruction.

I have already connected an LED (via a 220Ohm resistor) to GPIO 21. Ensure that the short pin of the LED is connected closer to the GND pin, and the longer pin closer to GPIO 21. You can place the resistor on either side of the LED, it really doesn't matter which side.



This LED is controlled by GPIO 21.

MicroPython contains a module that allows us to work with the file system. This module is named "uos". You can see its documentation for details.

Let's use it now.

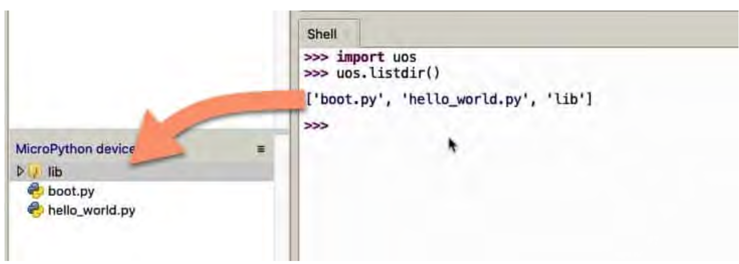
Start by importing the uos module so you can use its functions:

```
>>> import uos
```

Next, call the listdir() function which will list any files in the root directory:

>>> uos.listdir()

The result is this:



listdir() returns a tupple with the contents of the root directory.

There are two files ("hello_world.py" and "boot.py") and one directory ("lib") in the root directory of my ESP32's file system.

Say that you'd like to execute the program contained in the hello_world.py file. to do that, use Of course, the hello_world.py contains this script. And I can just type it in the Shell and run it interactively. I just say hello here and it will come back. Now, let's say that instead of you typing the interactive command into the Shell, you want to execute an existing file like the hello_world.py file, right? So, how do you do that?

You can use the execfile() function:

>>> execfile("hello_world.py")

The response is this:

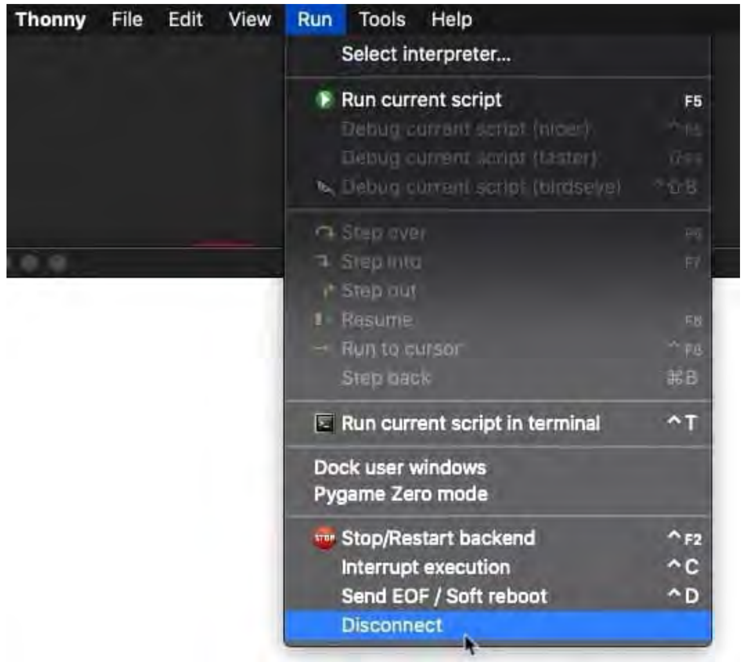
```
Shell
>>> import uos
>>> uos.listdir()
['boot.py', 'hello_world.py', 'lib']
>>> print("Hello")
  Hello
>>> execfile("hello_world.py")
  Hello World from MicroPython.
>>>
```

Use execfile() to execute a MicroPython program on the shell.

Now, let’s do the same experiment using the Serial tool, instead of Thonny.

Shell with Serial

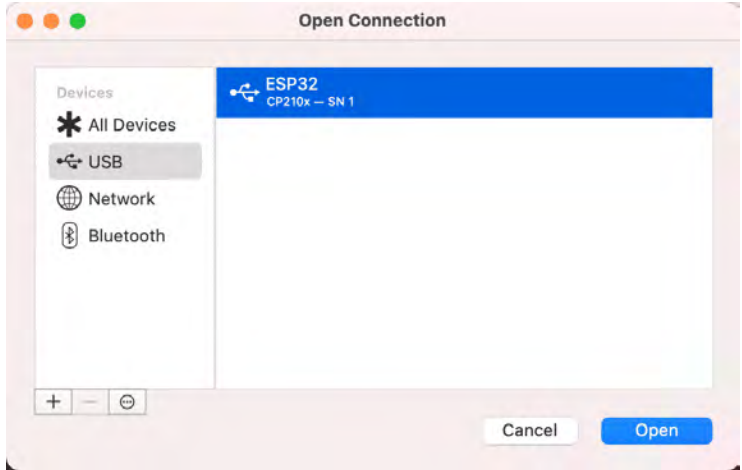
Only one client can access the Python or MicroPython shell at a time. Before you can use Serial, you must disconnect Thonny. You can do this by clicking on “Disconnect” under the Run menu.



Disconnect the ESP32 from Thonny to use a different shell utility.

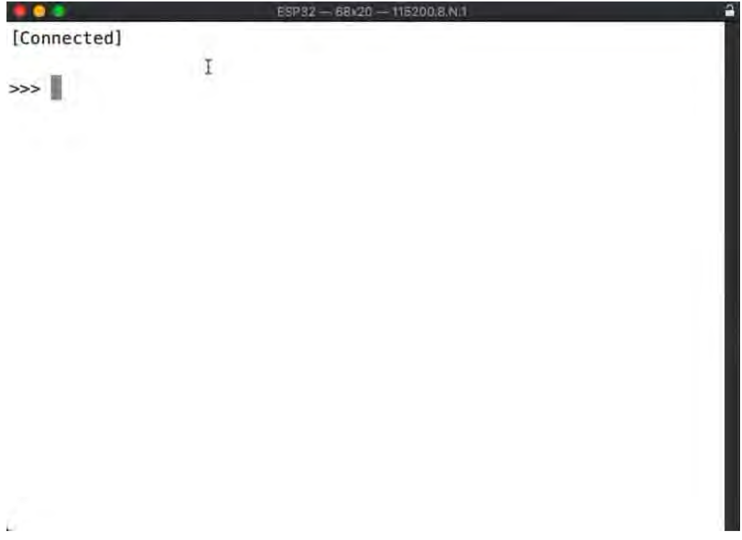
Next, start Serial and type Command-D (or Terminal –> Connect, via the menu) to connect to your ESP32.

In Serial you can bookmark a USB or network device so that you can connect to it quickly. In most cases, Serial will automatically detect and configure a USB serial port. Here's the bookmark for my ESP32:



The bookmark for my ESP32 in Serial.

Once Serial is connected to the MicroPython shell running on the ESP32, you will see the familiar MicroPythyon prompt (">>>"). MicroPython is ready for your instructions.



Serial is connected to the MicroPython shell.

From here, you can interact with the shell as you would in Thonny or any other tool. Let’s do the same experiment.

>>> import uos>>> uos.listdir()>>> execfile(“hello_world.py”)

Here is the shell session as it looks in Serial:

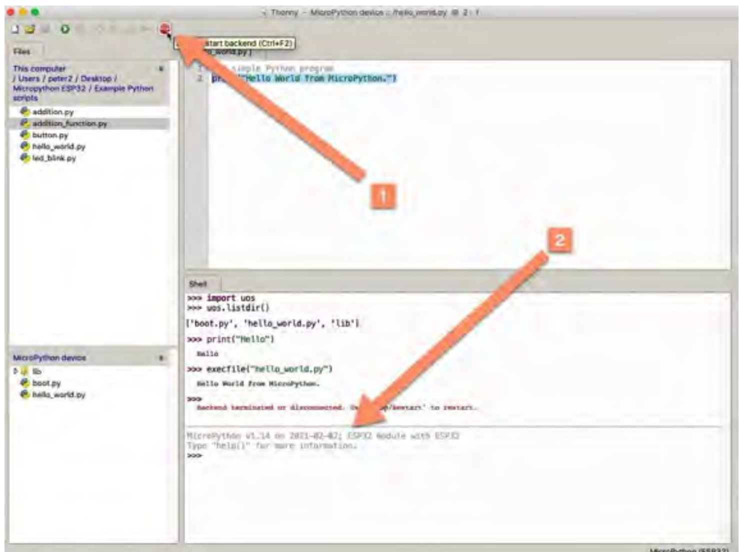
```
ESP32 — 88x20 — 115200,8,N,1
[Connected]

>>> import uos
>>> uos.listdir()
['boot.py', 'hello_world.py', 'lib']
>>> execfile("hello_world.py")
Hello World from MicroPython.
>>>
```

An example MicroPython shell session in Serial.

To disconnect Serial so you can use Thonny again, type Command-D or use the menu (Terminal –> Disconnect).

Shell with Thonny: LEDs and blocks

Back in Thonny, you can click on the Stop button (yes, I know, counter-intuitive but that’s how it is!). The ESP32 is now connected to Thonny.



The Stop buttons actually toggles connect/disconnect. When you reconnect, you will see a new shell prompt.

Let’s have a look at another example that involves the LED.

Copy this code into your MicroPython shell:

>>> from machine import Pin>>> led = Pin(21, Pin.OUT)>>> led.on()>>> led.off()>>> led.on()>>> led.off

Here’s how this looks in the shell:

```
Shell
>>> import uos
>>> uos.listdir()
['boot.py', 'hello_world.py', 'lib']
>>> print("Hello")
  Hello
>>> execfile("hello_world.py")
  Hello World from MicroPython.
>>>
  Backend terminated or disconnected. Use 'Stop/Restart' to restart.

MicroPython v1.14 on 2021-02-02; ESP32 module with ESP32
Type "help()" for more information.
>>> from machine import Pin
>>> led = Pin(21, Pin.OUT)
>>> led.on()
>>> led.off()
>>> led.on()
>>> led.off()
>>> |
MicroPython (ESP32)
```

Controlling an LED on the MicroPython shell.

I'll be explaining in detail what each of these instructions do in a dedicated part of the MicroPython course. In summary, I am creating an "led" object of class Pin, for the LED that is connected to GPIO 21, and use the on() and off() functions to control its state.

Easy, right?

The MicroPython shell also allows you to create blocks of code, complete with the ability to deal with Python indentations.

Here is one example (use the tab key or the space bar to enter the same indentation for each line inside the "while" block):

>>> from utime import sleep>>> while True: led.on() sleep(0.5) led.off() sleep(0.5)

The while block implements an infinite loop. In each cycle, the LED will turn on for 0.5 seconds and off for 0.5 seconds.

To exit the infinite loop, type Ctr-C.

```
Shell
  Hello world from MicroPython.

>>>
  Backend terminated or disconnected. Use 'Stop/Restart' to restart.

MicroPython v1.14 on 2021-02-02; ESP32 module with ESP32
Type "help()" for more information.
>>> from machine import Pin
>>> led = Pin(21, Pin.OUT)
>>> led.on()
>>> led.off()
>>> led.on()
>>> led.off()
>>> from utime import sleep
>>> while True:
        led.on()
        sleep(0.5)
        led.off()
        sleep(0.5)

  Traceback (most recent call last):
    File "<stdin>", line 5, in <module>
  KeyboardInterrupt:

>>> led.on()
>>> |
                                                MicroPython (ESP32)
```

I have used Ctrl-C to interrupt a running MicroPython program

The shell can remember ealier instructions. You can use the up and down arrow keys to navigate history, and hit the return to re-submit an old instruction.As you can see, the shell is easy to use, and provide a quick way to prototype and test instructions as you are working on a program.

Learn MicroPython for the ESP32

With this video course, you will learn how to use theMicroPython programming language with the ESP32 micro-controller.

MicroPython is the perfect language for anyone looking for the easiest (yet still powerful) way to program a micro-controller.

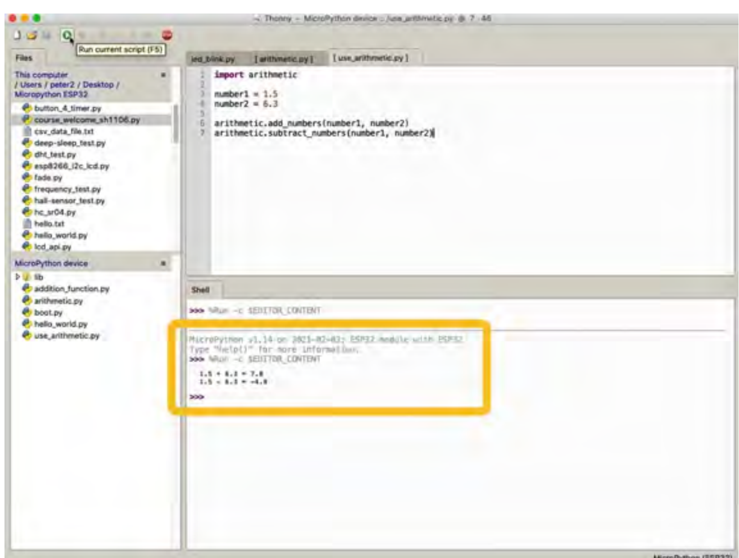
__CONFIG_colors_palette__{“active_palette”:0,”config”:{“colors”:{“3e1f8”:{“name”:”Main Accent”,”parent”:-1}},”gradients”:[]},”palettes”:[{“name”:”Default Palette”,”value”:{“colors”:{“3e1f8”:{“val”:”rgb(217, 49, 33)”}},”gradients”:[]},”original”:{“colors”:{“3e1f8”:{“val”:”rgb(19, 114, 211)”,”hsl”:{“h”:210,”s”:0.83,”l”:0.45}}},”gradients”:[]}}]}__CONFIG_colors_palette__

14. MicroPython Programming with files

MICROPYTHON WITH THE ESP32 GUIDE SERIES

MicroPython Programming With Files

In this lesson I will show you how to split your MicroPython programs into multiple files so that you can better organize your code.

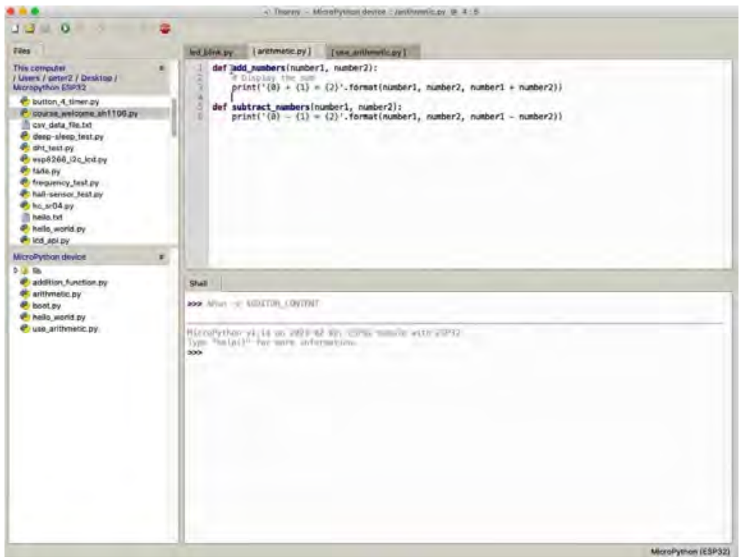


In this lesson I will show you how to split your MicroPython programs into multiple files so that you can better organize your code.

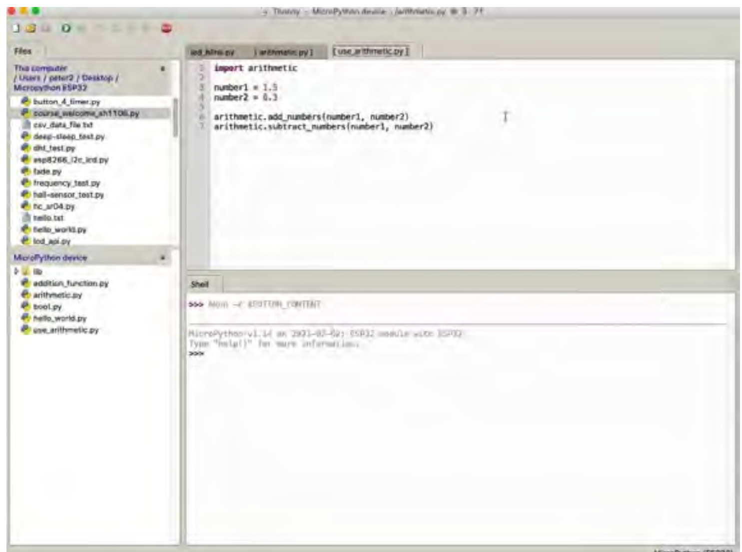
Doing so is particularly useful as your programs become larger and more complicated.

One program, two files

To show you how to split your MicroPython program into multiple files, I have prepared a simple example. The example contains two files, that I list below (first, screenshots that show how these files look in Thonny, followed by the code):



arithmetic.py in a Thonny tab.



use_arithmetic.py in a Thonny tab.

arithmetic.py

def add_numbers(number1, number2): # Display the sum print('{0} + {1} = {2}'.format(number1, number2, number1 + number2))

def subtract_numbers(number1, number2): print('{0} – {1} = {2}'.format(number1, number2, number1 – number2))

use_arithmetic.py

import arithmetic

number1 = 1.5number2 = 6.3

arithmetic.add_numbers(number1, number2)arithmetic.subtract_numbers(number1, number2)

arithmetic.py

The first file is titled “arithmetic.py”, and it contains two functions.

The first one is “add_numbers” that receives two numerical parameters, adds them, and prints the result to the shell.

The second function is titled “subtract_numbers”. This function also receives two parameters, subtracts them and prints out the result to the shell.Don’t worry about the details of how these programs work. I have a complete section in the course dedicated to the MicroPython language where I explain everything, including the use of the format function, how to create custom functions, parameters and classes.

use_arithmetic.py

The second file is titled “use_arithmetic.py”.

In the first line, we use the import keyword to import the contents of the “arithmetic.py” file. With this import, we can now use the two functions defined inside arithmetic.py without having to write them again.

In other words, the file arithmetic.py consists of a small MicroPython module that we can import and use in other programs.

To import a module, we use the import keyword followed by the name of the file that contains the code that we want to import, without the “.py” extension.

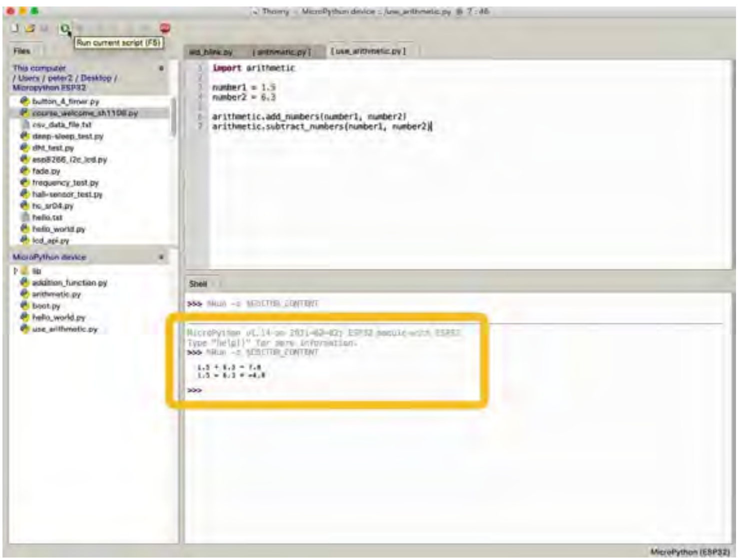
Next in use_arithmetic.pu, I create two variables to hold the numbers that I want to use later, and give each an initial value.

Finally, I call the calculation functions that exist in the arithmetic module to do the addition and subtraction between

those numbers.

Execute the program

As with any MicroPython program in Thonny, to run it, simply click on the play button.



My modular program has executed. Output is in the shell.

Why split a program?

The files in this example are tiny. But imagine your programs getting bigger and bigger as you become more proficient in the language. Being able to split them into multiple files results in programs that are more efficient, easier to manage, easier to modify and easier to share. Each file can be dedicated to a single task or a set of closely related tasks.

Learn MicroPython for the ESP32

With this video course, you will learn how to use theMicroPython programming language with the ESP32 micro-controller.

MicroPython is the perfect language for anyone looking for the easiest (yet still powerful) way to program a micro-controller.

__CONFIG_colors_palette__{“active_palette”:0,”config”:{“colors”:{“3e1f8”:{“name”:”Main Accent”,”parent”:-1}},”gradients”:[]},”palettes”:[{“name”:”Default Palette”,”value”:{“colors”:{“3e1f8”:{“val”:”rgb(217, 49, 33)”}},”gradients”:[]},”original”:{“colors”:{“3e1f8”:{“val”:”rgb(19, 114, 211)”,”hsl”:{“h”:210,”s”:0.83,”l”:0.45}}},”gradients”:[]}}]}__CONFIG_colors_palette__

15. How to interrupt a running program

MICROPYTHON WITH THE ESP32 GUIDE SERIES

How To Interrupt A Running Program

In this lesson, I’ll show you how to interrupt a running program and how to restart your ESP32 using soft reset.

To demonstrate how to interrupt a running program and rest your ESP32, I’ll use our familiar “blinking LED script”.

Here is the script:

from machine import Pinfrom utime import sleepled = Pin(21, Pin.OUT) while True: print(“.”) led.on() sleep(0.5) led.off() sleep(0.5)

Notice that inside the “while” block, I have added a line that prints a dot character in the shell. This will help us recognise that the program is running without having to look at the LED on the breadboard.In this lesson, we’ll take a closer look at the

boards that can use MicroPython.

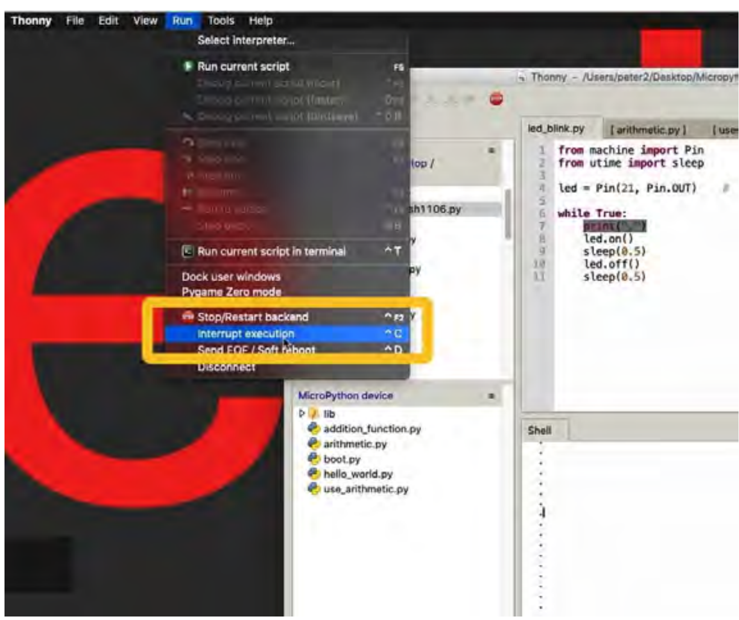
Ctrl-C to interrupt a program

Click on the play button to start the program. Notice that the LED is blinking, and a new dot character appears in the shell every 0.5 seconds.

While the program is running, you cannot use the shell. To re-gain access to the shell, you must interrupt the program. This, essentially, will end the program execution.

To interrupt the program you can use the Interrupt execution command, under the Run menu.

Or, you can type Crtl-C on your keyboard.



Control-C will interrupt a running program and give you access to the shell.

```
.
.
.
.
.
.
.
.
.
.
.
Traceback (most recent call last):
  File "<stdin>", line 11, in <module>
KeyboardInterrupt:
>>>
```

MicroPython (ESP32)

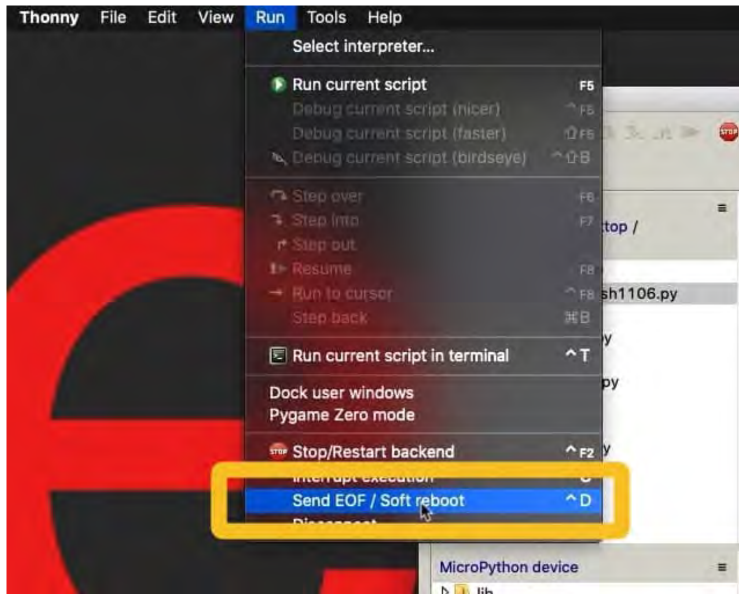
Pressing Ctrl-C results to a KeyboardInterrupt event.

Ctrl-D to soft-reboot a program

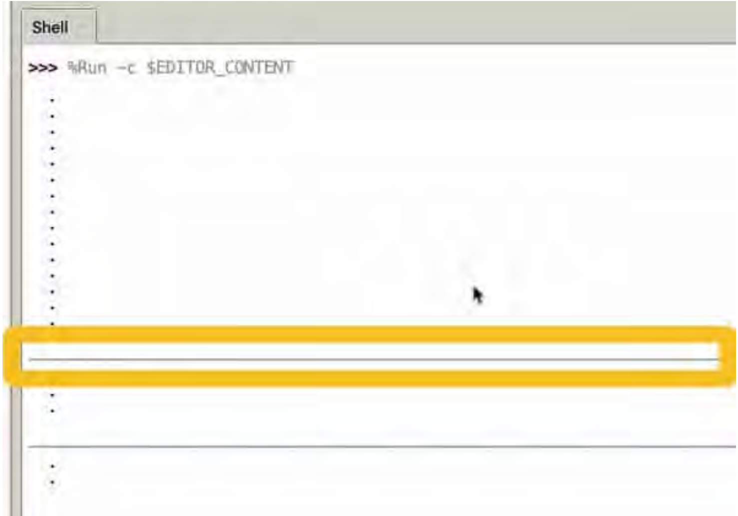
As opposed to a keyboard interrupt via Ctrl-C, you can do a soft-reboot by typing Ctrl-D, or selecting “Send EOF / Soft Reboot” from the Run menu.

When you do a soft reboot, your ESP32 restarts your program from “scratch”. As opposed to Ctrl-C, you do not get access to the shell, since the program is still running.

Try Ctrl-D while the blinking LED program is running. Notice that after you do a soft-reboot, the shell shows a horizontal line, and then new dots appear, generated by the running program.



Ctrl-D does a soft-reboot.



The soft-reboot is indicated by a horizontal line in the shell.

After a soft reboot, the RAM of your ESP32 will not be cleared. Any variable values remain in the RAM.

Learn more about the reset and boot modes in MicroPython here.

Stop/Restart backend

Another option available to us is to stop and restart the backend. The result of this is that your ESP32 does a “hard reset”. This is similar to pressing the reset button on the board itself.

Of course, a hard reset will stop the program and give you a fresh shell prompt. All RAM contents are lost, but the flash memory remains so any files stored in the ESP32 file system will remain intact.

You can also do a hard-reset by clicking on the Stop button in Thonny.

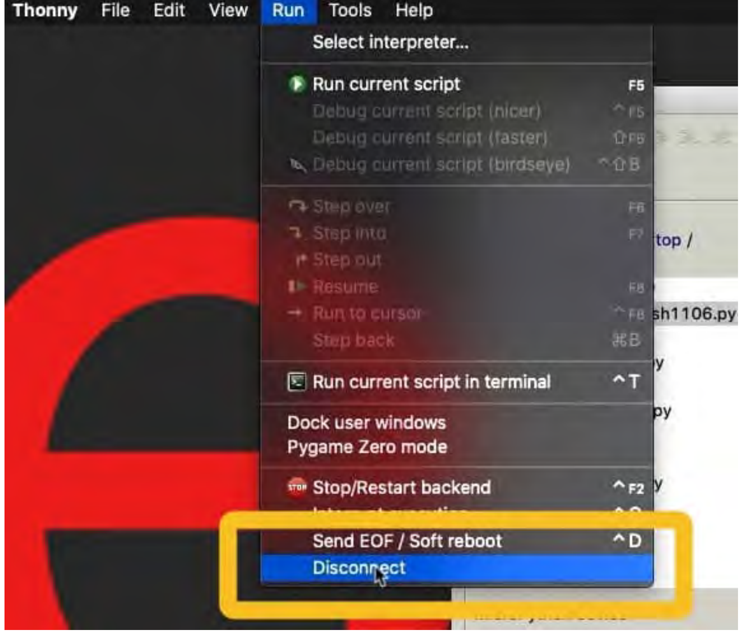
Stop/Restart backend has the effect of a hard-reset operation.

```
>>>
MicroPython v1.14 on 2021-02-02; ESP32 module with ESP32
Type "help()" for more information.
>>>
```

After a hard-reset, a running program will stop and you will get a fresh shell command prompt.

Disconnect

Finally, you can simulate the physical disconnection and connection of the USB cable by selecting Disconnect from the Run menu.



Edit your caption text here

Often, this is an effective way to deal with communications problems between your computer and the target device. If a software disconnect and connect does not help, the next step would be to physically disconnect and connect the USB cable.

Learn MicroPython for the ESP32

With this video course, you will learn how to use theMicroPython programming language with the ESP32 micro-controller.

MicroPython is the perfect language for anyone looking for the easiest (yet still powerful) way to program a micro-controller.

__CONFIG_colors_palette__{“active_palette”:0,”config”:{“colors”:{“3e1f8”:{“name”:”Main Accent”,”parent”:-1}},”gradients”:[]},”palettes”:[{“name”:”Default Palette”,”value”:{“colors”:{“3e1f8”:{“val”:”rgb(217, 49, 33)”}},”gradients”:[]},”original”:{“colors”:{“3e1f8”:{“val”:”rgb(19, 114, 211)”,”hsl”:{“h”:210,”s”:0.83,”l”:0.45}}},”gradients”:[]}}]}__CONFIG_colors_palette__

16. How to run a program at boot

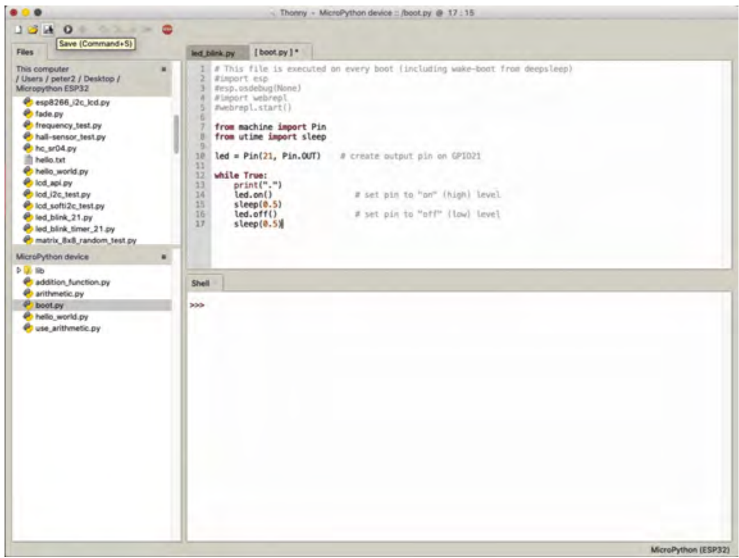
MICROPYTHON WITH THE ESP32 GUIDE SERIES

How To Run A Program At Boot

In this lesson, I’ll show you how to set your ESP32 to execute a program when it powers up or when you press the reset button.

To make this possible, we’ll use the boot.py and main.py files.

I will also show you how to stop autostart so you can regain control of your ESP32.



I updated this guide page on October 6, 2023. In this update I added information about the “main.py” file, on how to stop

“boot.py” from hijacking your ESP32. I will update the video shortly.

Up to now, you have been programing the ESP32 by having it constantly connected to the computer. From Thonny, you click on the green “play” button to trigger the selected program to run.

Imagine that you have finished working on your program, and want your ESP32 to be independent of your computer. You want the ESP32 and to be able to automatically execute a given program when power is applied.There are a couple of ways by which you can do that.

In this lesson, I will show you both of them.

In the first method, we’ll use the “boot.py” file. In the second method, we’ll use the “main.py” file. Before we start, let’s understand the differences and similarities between boot.py and main.py.

In this lesson, I will show you both of them.

In the first method, we’ll use the “boot.py” file. In the second method, we’ll use the “main.py” file. Before we start, let’s understand the differences and similarities between boot.py and main.py.

boot.py vs main.py

When you’re working with MicroPython, you’ll often come across two important files: boot.py and main.py. These files serve distinct purposes and understanding their roles can help you structure your projects more effectively.

Think of boot.py as the welcoming committee for your MicroPython board. As soon as the board powers up or resets, boot.py is the first file that gets executed. This file is your go-to place for setting up initial configurations like network

settings or initializing peripherals. It’s crucial to keep this file as minimal as possible. The reason is simple: if something goes wrong in boot.py, it could prevent the board from booting up properly, and you might not even get to the point where main.py runs.

Once boot.py has done its job, main.py takes over. This file is where the meat of your application resides. Whether you’re running a web server, reading from sensors, or executing any other main tasks, main.py is where all this action happens. Unlike boot.py, which runs just once at boot-up, main.py runs in a loop, continuously executing its code unless you’ve programmed it to do otherwise.

The Order of Execution

The sequence is straightforward. Boot.py runs first, setting the stage for main.py, which follows immediately after. If boot.py is the opening act, main.py is the main event.

Handling Errors

It’s worth noting that errors in boot.py can stop main.py from running. On the other hand, if something goes awry in main.py, it won’t affect the initial boot process, which is managed solely by boot.py.

Flexibility and Control

Both boot.py and main.py are accessible via the file system, meaning you can edit or even remove them as you see fit. However, it’s generally a good idea to exercise caution, especially with boot.py, to ensure a smooth boot-up process.

As you can see, you can add code to both files. Generally, you’ll add initialisation code to boot.py, and the main application code to main.py. But in simple scenarios, such as getting an LED to blink, you can add your code to either file. That’s what we’ll do next.

boot.py

In MicroPython, “boot.py” is a special-purpose file.

If it exists in the root of the MicroPython filesystem, the ESP32 will try to read it and execute the program it contains.

led_blink.py   [boot.py]

```
# This file is executed on every boot (including wake-boot from deepsleep)
#import esp
#esp.osdebug(None)
#import webrepl
#webrepl.start()
```

The default boot.py file. Notice that the program it contains is commented out.

When you install a fresh instance of the MicroPython firmware on your ESP32 or other hardware target, a boot.py file will be written in the filesystem. You can open this file in Thonny to have a look. As you can see in the screenshot above, the default boot.py file does contain some code, but it is commented out, and therefore inactive.

You can replace this code with yours.

Let’s do that now.

Experiment 1: using boot.py

To demonstrate how boot.py works, I’ll use the sample code from the previous lecture.

Here’s the code:

from machine import Pinfrom utime import sleepled = Pin(21, Pin.OUT) while True: print(“.”) led.on() sleep(0.5) led.off() sleep(0.5)

Copy this code in the boot.py file so that it looks like this (I did not replace the existing commented-out code):

```
led_blink.py   [ boot.py ] *

1  # This file is executed on every boot (including wake-boot from deepsleep)
2  #import esp
3  #esp.osdebug(None)
4  #import webrepl
5  #webrepl.start()
6
7  from machine import Pin
8  from utime import sleep
9
10 led = Pin(21, Pin.OUT)    # create output pin on GPIO21
11
12 while True:
13     print(".")
14     led.on()              # set pin to "on" (high) level
15     sleep(0.5)
16     led.off()             # set pin to "off" (low) level
17     sleep(0.5)
```

The new content of the boot.py file.

Now that this code exists in the boot.py file, it will execute automatically next time I power-up the device. I don’t need to connect the device to the computer, and I don’t need to click on the play button in Thonny to run the program.

To test this, follow this process:

1. Disconnect the device from the computer by unplugging the USB cable from the computer USB port (leave the cable connected to the device).
2. Connect the USB cable to a USB power supply, like a phone charger. This will power up the device.
3. The program stored in boot.py will start , and the LED connected to GPIO 21 will start

to blink.

What you have achieved is to make your ESP32 independent of your computer.

Problem: boot.py takes over your ESP32 (and how to deal with this)

Because the example code in boot.py contains an infinite loop, when you reconnect the ESP32 to your computer so that you can continue your work in Thonny, the ESP32 will be busy blinking the LED and will not give you a MicroPython shell, allow you to modify the program, or interact with the filesystem. Essentially, your ESP32 is locked up in the infinite loop.

Normal ways of interrupting the program will not work. Clicking on the Stop button or typing Ctrl-C will simply reboot the board, and restart execution of the program in the boot.py file.

The only way (that I am aware off) for regaining control of your board so that you can make changes to the boot.py program (or to delete it) is to re-flash the board. Follow the method you learned in this earlier article in this series.

Experiment 2: import a program into boot.py

When you have a small program to run at boot up, copying it into the boot.py file is not a problem; it just works.

Imagine that you have a larger and more complicated program. You’ve been working on it for a while, and it works well enough to deploy in the field.

Copying a large file into boot.py is an option but it is messy.

You will end up with two copies of the same file. If you need to make changes, you'll have to remember to apply the changes to both files.

That's a recipe for problems down the track that is easily avoided using an import statement.

You have already learned how to use the import statement.

In this example, the program that you want to execute in boot up is stored in the file titled "led_blink.py".

Inside boot.py, simply use this line of code:

import led_blink

At boot up, the ESP32 will read boot.py which contains the import statement. Then, the ESP32 will import the code stored in led_blink.py and execute it. The result will be, predictably, a blinking LED.

led_blink.py [boot.py]

```
1 # This file is executed on every boot (including wake-boot from
2 #import esp
3 #esp.osdebug(None)
4 #import webrepl
5 #webrepl.start()
6
7 import led_blink
```

The boot.py file contains an import statement for the main program file.

To test that this method works, repeat the same 3 steps as listed in Experiment 1 in this lesson. Power up the ESP32 from an external power supply (not your computer), to confirm that

the LED on GPIO21 is blinking.

main.py

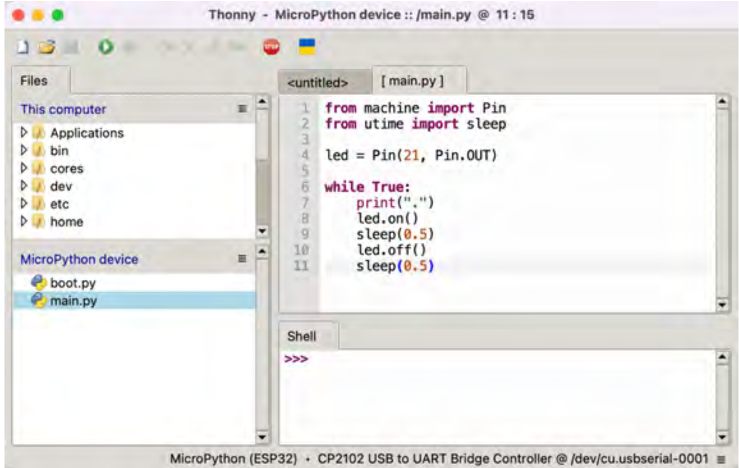
As you learned earlier, in Micropython, there is the main.py file that is executed automatically after boot.py. If your program is simple, and does not need any special code such as boot-up configuration, you can use main.py instead of boot.py to automatically execute a program when your board starts.

In almost every respect, boot.py and main.py do the same thing: they automatically execute a program at startup. However, with main.py, you can easily stop execution with Ctrl-C instead of having to re-flash the board. Therefore, main.py is easier, and more appropriate in most use cases.

Let’s repeat the LED-blink experiment with main.py. In Thonny, copy the following program into a new file with name “main.py”, and save this file on the device.

from machine import Pinfrom utime import sleepled = Pin(21, Pin.OUT)

while True: print(“.”) led.on() sleep(0.5) led.off() sleep(0.5)



The program in main.py

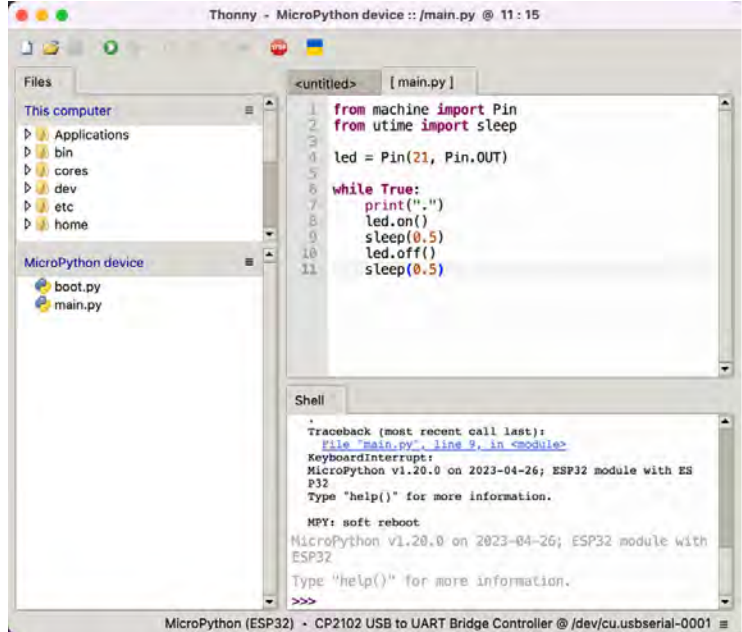
Just like with boot.py, after you save the main.py file, power-cycle the board (i.e. unplug the USB cable, and then plug it back into a USB port). When the board is powered again, the LED will blink, indicating that the code in the main.py file is running.

To stop the program, follow this process:

1. click on the STOP button in Thonny (this will connect the Thonny shell to the board),

2. Press and hold Ctrl-C until you see the “soft reboot” message.

At that time, the program in main.py will stop, you will re-gain control of the shell, and you will see the files in the device file system.



After clicking STOP and hold Ctr-C until a soft reboot occurs, you are back in control of your board.

Once you are back in control of your board, you can make any changes you want to the contents of the main.py file. Next time you power-cycle the board, the program in main.py will automatically start.

If you don't want to auto-start, simply rename or remove the main.py and boot.py files.

With the help of the boot.py and main.py files you can automatically run a program when your ESP32 is powered up.

The next thing that I want to show you is how to do simple debugging of your MicroPython scripts using Thonny IDE. Let's do that in the next lecture.

Learn MicroPython for the ESP32

With this video course, you will learn how to use theMicroPython programming language with the ESP32 micro-controller.

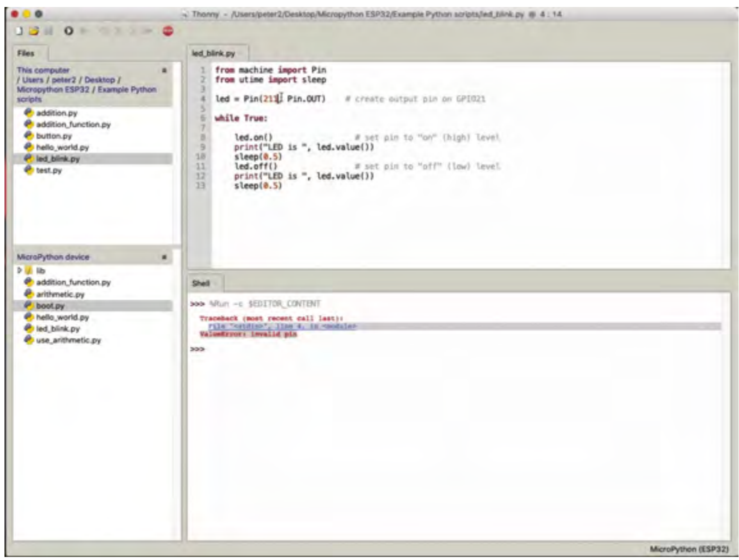
MicroPython is the perfect language for anyone looking for the easiest (yet still powerful) way to program a micro-controller.

__CONFIG_colors_palette__{“active_palette”:0,”config”:{“colors”:{“3e1f8”:{“name”:”Main Accent”,”parent”:-1}},”gradients”:[]},”palettes”:[{“name”:”Default Palette”,”value”:{“colors”:{“3e1f8”:{“val”:”rgb(217, 49, 33)”}},”gradients”:[]},”original”:{“colors”:{“3e1f8”:{“val”:”rgb(19, 114, 211)”,”hsl”:{“h”:210,”s”:0.83,”l”:0.45}}},”gradients”:[]}}]}__CONFIG_colors_palette__

MICROPYTHON WITH THE ESP32 GUIDE SERIES

How To Debug A MicroPython Program

In this lesson, I will show you a few techniques you can use to debug and troubleshoot your MicroPython scripts.



To demonstrate how to debug and troubleshoot MicroPython scripts using the Thonny editor, we'll use the same demo script that your are familiar from the previous few lectures.

Here the script:

from machine import Pinfrom utime import sleepled = Pin(21, Pin.OUT) while True: print(".") led.on() sleep(0.5) led.off()

sleep(0.5)

Use print statements to help with tracing

If you come from the Arduino world, then you are familiar with how we use print statements there to try and figure out what is happening during runtime.

In the screenshot I provide below, I have added a couple of print statements that contain a value that I am interested in using to evaluate whether my program is working properly or not. The print statement looks like this:

print(“LED is “, led.value())

In the screenshot, you can see the output of these print statements in the shell.

Why are print statements (like in this example) useful?

Here’s a simple scenario:

Imagine that I had not properly connected the LED to GPIO21, or the LED was damaged.

Even though this program is correct, the LED would not blink. By including the led value in the print statement, I have confirmation in the shell that my program is working properly.

This would prompt me to look at the LED for the source of the problem (i.e. the LED not blinking as expected).

Of course, this is a simple scenario. However, print statements can scale up to much more complicated situations so they comprise of a indispensable debugging tool.

led_blink.py

```
from machine import Pin
from utime import sleep

led = Pin(21, Pin.OUT)    # create output pin on GPIO21

while True:

    led.on()               # set pin to "on" (high) level
    print("LED is ", led.value())
    sleep(0.5)
    led.off()              # set pin to "off" (low) level
    print("LED is ", led.value())
    sleep(0.5)
```

Shell

```
>>> %Run -c $EDITOR_CONTENT
  LED is  1
  LED is  0
  LED is  1
  LED is  0
  LED is  1
  LED is  0
  LED is  1
  LED is  0
  LED is  1
```

Print statements provide a simple way to trace the execution of your program by printing out debug values in the shell.

MicroPython error messages

The MicroPython interpreter produces error messages that can provide valuable clues for when things go wrong. Even though some times the information they provide can be very broad and generic, in most cases you can use these messages to find and fix a programming bug within seconds.

Example 1: parameter typo bugs

Let’s have a look at a simple example.

In line 4 of the demo program, I have introduced an innocent

type. It can happen to all of us. Here’s the new “buggy” line 4 (bug in bold):

led = Pin(21**1**, Pin.OUT)

This bug results in MicroPython trying to setup a GPIO that does not exist.

When you try to execute this file, you will see a “ValueError” in the shell.

It looks like this:

This typo has generated a very specific and useful error message.

The error message is very useful because it is very specific. It both identifies line 4 as the location of the bug, and even the kind of but that it is (“invalid pin”).

Fixing it is quick and easy, just remove the redundant “1”.

Let’s look at one more bug that involves an incorrect parameter. Here’s another buggy line, again in line 4 of the demo program (bug in bold):

led = Pin(21, Pin.OUT**s**)

Run the program. The interpreter complains an “AttributeError”.

Again, this error is very specific, and informs you that the problem is in line 4, and the the object “Pin” does not have an attribute “OUTs”:

```
led_blink.py
1  from machine import Pin
2  from utime import sleep
3
4  led = Pin(21, Pin.OUTs)    # create output pin on GPIO21
5
6  while True:
7
8      led.on()                # set pin to "on" (high) level
9      print("LED is ", led.value())
10     sleep(0.5)
11     led.off()               # set pin to "off" (low) level
12     print("LED is ", led.value())
13     sleep(0.5)

Shell
>>> %Run -c $EDITOR_CONTENT

  Traceback (most recent call last):
    File "<stdin>", line 4, in <module>
  ValueError: invalid pin

>>> %Run -c $EDITOR_CONTENT

  Traceback (most recent call last):
    File "<stdin>", line 4, in <module>
  AttributeError: type object 'Pin' has no attribute 'OUTs'

>>>
```

This error message specifically highlights the incorrect attribute.

With this information, you can easily find the bug and fix it.

Generally, bugs that involve parameters produce accurate and descriptive error messages. But not all error messages are like this. Let's look at an example where a simple typo generates an error message that is not as accurate.

Example 2: Name errors

A "NameError" is an error that relates to using an incorrect keyword. A keyword can be something like a variable or a reserved language keyword.

For example, in the demo program, I will introduce a bug by making a typo in the "while True:" loop statement. After the bug, the relevant line looks like this (bug in bold):

while Tru**a**e:

When you try to execute the program, a "NameError" error message is generated:

```
led_blink.py

1  from machine import Pin
2  from utime import sleep
3
4  led = Pin(21, Pin.OUT)    # create output pin on GPIO21
5
6  while Truae:
7
8      led.on[illegible]             # set pin to "on" (high) level
9      print("LE[illegible] ", led.value())
10     sleep(0.5)
11     led.off()                # set pin to "off" (low) level
12     print("LED is ",[illegible]d.value())
13     sleep(0.5)
```

```
Shell

>>> %Run -c $EDITOR_CONTENT

  Traceback (most recent call last):
    File "<stdin>", line 4, in <module>
  ValueError: invalid pin

>>> %Run -c $EDITOR_CONTENT

  Traceback (most recent call last):
    File "<stdin>", line 4, in <modu[illegible]
  AttributeError: type object 'Pin'[illegible]o attribute 'OUTs'

>>> %Run -c $EDITOR_CONTENT

  Traceback (most recent call last):
    File "<stdin>", line 13, in <module>
  NameError: name 'Truae' isn't defined

>>>
```

The bug is in line 6, but the error message indicates line 13.

This bug is more challenging to fix because it indicates the location of the bug in line 13, even though the actual location is line 6.

However, the clue that is more useful is that the buggy keyword “Truae” is included in the error message. In such cases, you can do a quick text search in your program for the offending keyword so that you can find it, and then fix it.Lesson to remember: error messages provide clues that you have to consider in their entirety. The line number is just one of these clues, and it may be incorrect.

Example 3: Positional arguments

Let’s look at one more bug. In one of the led functions, I have introduced an incorrect parameter, like this (bug in bold):

led.off(**3**)

As you probably know, the Pin.on() and Pin.off() functions do not take any arguments. When you try to run this program, the interpreter will throw a “TypeError”, like this:

led_blink.py

```
1  from machine import Pin
2  from utime import sleep
3
4  led = Pin(21, Pin.OUT)    # create output pin on GPIO21
5
6  while True:
7
8      led.on()              # set pin to "on" (high) level
9      print("LED is ", led.value())
10     sleep(0.5)
11     led.off(3)            # set pin to "off" (low) level
12     print("L[illegible] is ", led.value())
13     sleep(0.5[illegible]
```

Shell

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

  Traceback (most recent call last):
    File "<stdin>", line 4, in <module>
  ValueError: invalid pin

>>> %Run -c $EDITOR_CONTENT

  Traceback (most recent call last):
    File "<stdin>", line 4, in <module>
  AttributeError: type object 'Pin' has no attribu[illegible] 'OUTs'

>>> %Run -c $EDITOR_CONTENT

  Traceback (most recent call last):
    File "<stdin>", line 13, in <module>
  NameError: name 'Truae' isn't defined

>>> %Run -c $EDITOR_CONTENT

  LED is  1
  Traceback (most recent call last):
    File "<stdin>", line 11, in <module>
  TypeError: function takes 1 positional arguments but 2 were given

>>>
```

This error message identifies the correct bug location, but the description is confusing

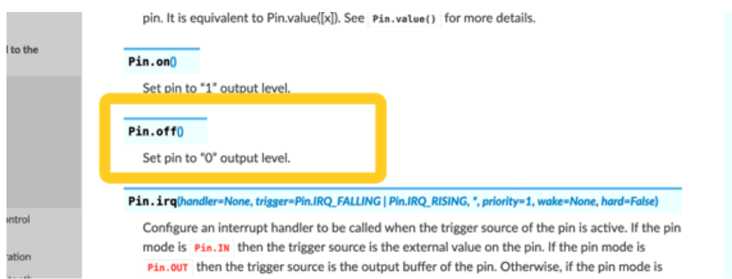
This error message identifies the correct location of the bug, line 11.

However, the description is confusing. The description suggests that the function “off” requires one positional argument, but that two arguments were given. As you can clearly see, I have only given a single argument (not two, as the message claims), and I know that the correct number of arguments is zero (not one, as the message claims).

The counting of positional arguments in MicroPython functions relate to the “self” keyword, and I address it in a dedicated lecture in the course. But for now, it is worth taking into account the fact that this specific message can really throw you of and cause confusion, at least for a short time.

To fix it, I would start by not taking much note of the specific number of arguments that the error message indicates. Instead, I would look at the documentation of the specific function in question. This would give me authoritative information about how many arguments the function requires (if at all it requires arguments), and of what kind.

You can find the documentation for the off() function here. As you can see, this function takes no arguments.



The Pin.off() function takes no arguments.

With this information you can reason that the problem is the value “3” in the off() function. You’ll be able to fix the bug by removing the “3”.

In the course, during the experiments, we’ll be bumping into various programming problems. And in many cases, I’ll be showing you “live” how I solved those problems.

But for now, keep in mind what you have learned in this lessons as it will help you save a lot of time and spare a lot of frustration as you are programming your ESP32 using MicroPython.

Learn MicroPython for the ESP32

With this video course, you will learn how to use theMicroPython programming language with the ESP32 micro-controller.

MicroPython is the perfect language for anyone looking for the easiest (yet still powerful) way to program a micro-controller.

__CONFIG_colors_palette__{“active_palette”:0,”config”:{“colors”:{“3e1f8”:{“name”:”Main Accent”,”parent”:-1}},”gradients”:[]},”palettes”:[{“name”:”Default Palette”,”value”:{“colors”:{“3e1f8”:{“val”:”rgb(217, 49, 33)”}},”gradients”:[]},”original”:{“colors”:{“3e1f8”:{“val”:”rgb(19, 114, 211)”,”hsl”:{“h”:210,”s”:0.83,”l”:0.45}}},”gradients”:[]}}]}__CONFIG_colors_palette__