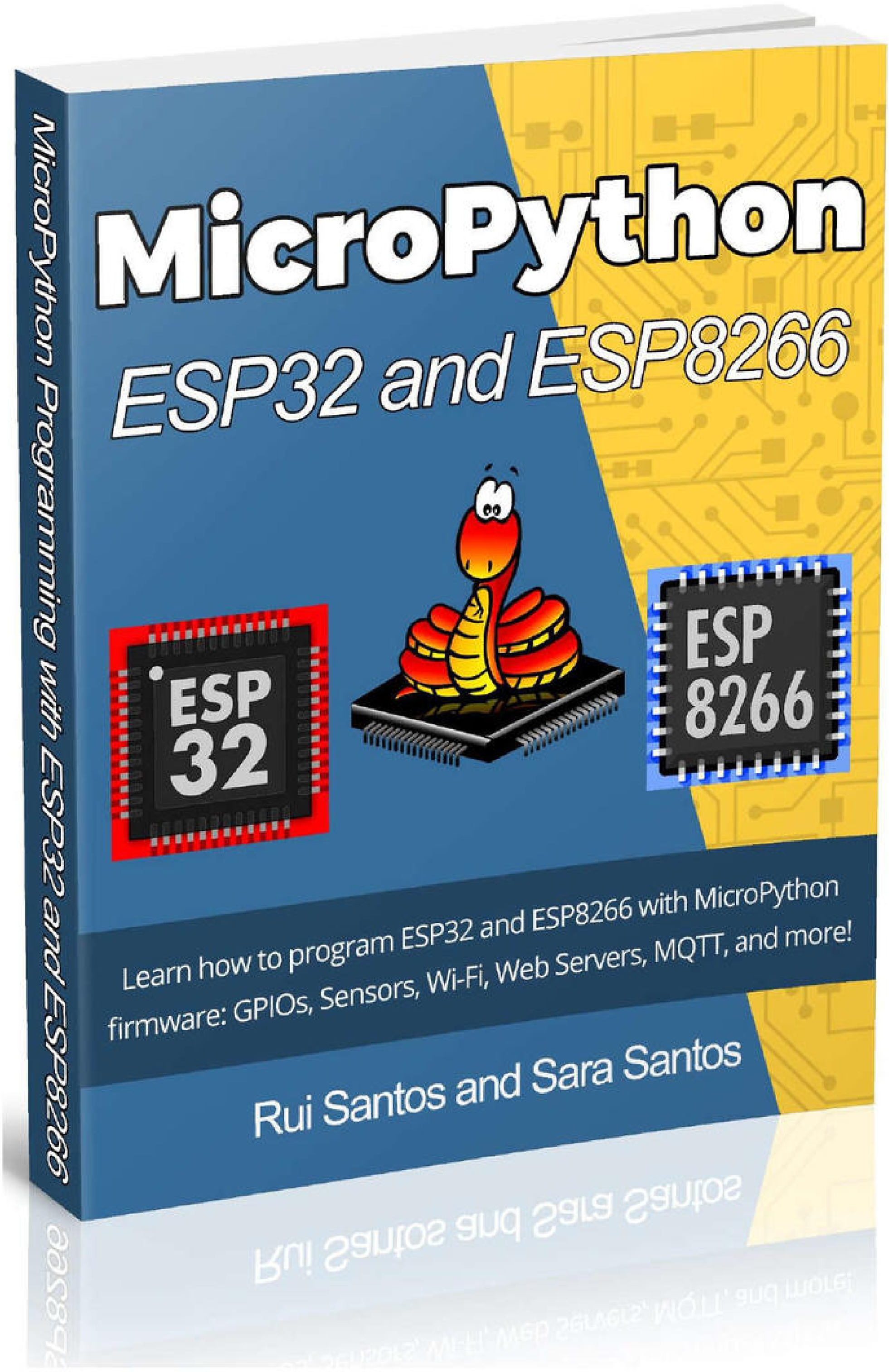

MicroPython
ESP32 and ESP8266
ESP 32
ESP 8266
Learn how to program ESP32 and ESP8266 with MicroPython firmware: GPIOs, Sensors, Wi-Fi, Web Servers, MQTT, and more!
Rui Santos and Sara Santos
MicroPython Programming with ESP32 and ESP8266

Version 1.2

Disclaimer

This eBook was written for information purposes only. Every effort was made to make this eBook as complete and accurate as possible. The purpose of this eBook is to educate. The authors (Rui Santos and Sara Santos) do not warrant that the information contained in this eBook is fully complete and shall not be responsible for any errors or omissions.

The authors (Rui Santos and Sara Santos) shall have neither liability nor responsibility to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by this eBook.

Throughout this eBook you will find some links and some of them are affiliate links. This means the authors (Rui Santos and Sara Santos) earn a small commission from each purchase with that link. Please understand that the authors have experience with all those products, and we recommend them because they are useful, not because of the small commissions we made if you decide to buy something. Please do not spend any money on these products unless you feel you need them.

Other Helpful Links:

- Ask questions in our Forum
- Join Private Facebook Group
- Terms and Conditions

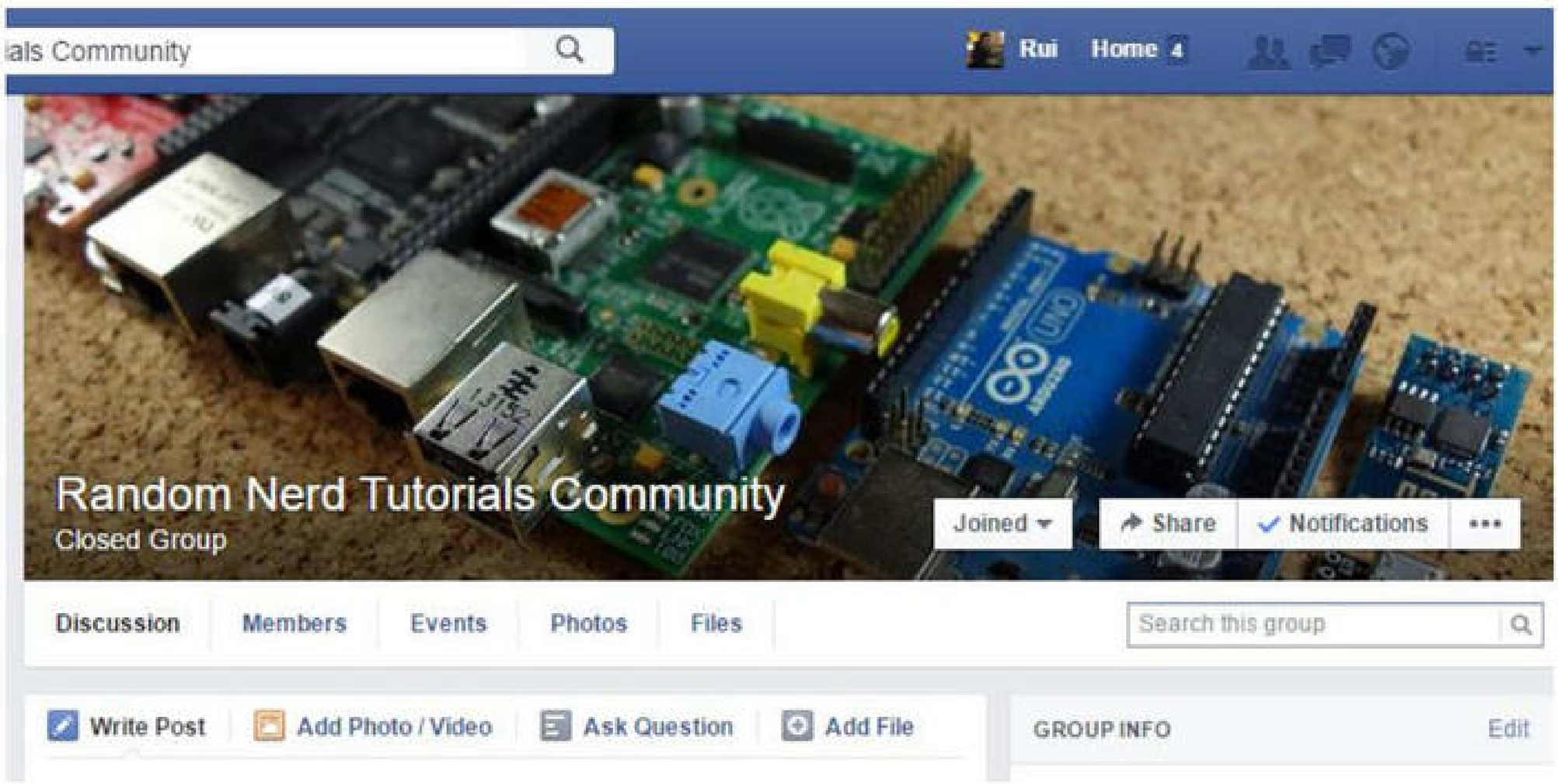
Join the Private Facebook Group

This course comes with an opportunity to join a private community of like-minded people. If you purchased this course, you can join our private Facebook Group today!

Inside the group, you can ask questions and create discussions about everything related to ESP32, ESP8266, Arduino, Raspberry Pi, BeagleBone, etc.

See it for yourself!

- Step #1: Go to -> http://randomnerdtutorials.com/fb
- Step #2: Click "Join Group" button
- Step #3: We'll approve your request within less than 24 hours.



About the Authors

This course was developed and written by Rui Santos and Sara Santos. We both live in Porto, Portugal, and we know each other since 2009. If you want to learn more about us, feel free to read our about page.

> *Hi! I'm Rui Santos, the founder of the Random Nerd Tutorials blog. I have a master's degree in Electrical and Computer Engineering from FEUP. I'm the author of "BeagleBone For Dummies", and Technical reviewer of the book "Raspberry Pi For Kids For Dummies". I recently wrote a book with Sara Santos for the NoStarchPress publisher about projects with the Raspberry Pi: "20 Easy Raspberry Pi Projects: Toys, Tools, Gadgets, and More!"*

> *Hi! I'm Sara Santos! I started working at Random Nerd Tutorials back in 2015 as a hobby: I helped Rui with some simple tasks when he had a lot of work to do. Back then, I knew nothing about electronics, programming, Arduino, etc... Over time, I started learning everything I could about those subjects and I just loved it! At first, I helped Rui once a week on Saturdays, but then, I started working on the RNT blog alongside him, almost every day. Currently, I work full time at Random Nerd Tutorials as a Content Editor and I love what I do!*

Table of Contents

MODULE 0

Course Intro 9

Welcome to MicroPython Programming with ESP32 and ESP8266 10

MODULE 1

Getting Started with MicroPython on ESP32 and ESP8266 12

Introducing MicroPython 13

Installing uPyCraft IDE (Windows) 18

Install uPyCraft IDE (Mac OS X) 23

Install uPyCraft IDE (Linux Ubuntu) 29

Flashing MicroPython Firmware to ESP32/ESP8266 32

Getting Started with uPyCraft IDE 45

Introducing the ESP32 Board 59

Introducing the ESP8266 Board 64

MODULE 2

Python/MicroPython Programming Basics 68

MicroPython Programming Basics 69

MODULE 3

Interacting with GPIOs 83

Blinking an LED 84

Digital Inputs and Digital Outputs 91

Analog Inputs 96

PWM – Pulse Width Modulation 104

ESP32 Touch Pins 116

Interrupts 120

Timers 129

Deep Sleep with Timer Wake Up 135

MODULE 4

Web Servers and HTTP Clients 143

Web Server Introduction 144

"Hello, World!" Web Server 152

Web Server – Control Outputs 163

Web Server with Slider Switch 171

Web Server - Display Temperature and Humidity Readings 179

Send Sensor Readings via Email (IFTTT) 192

Getting Data Using an API 212

MODULE 5

MQTT Protocol 221

Introducing MQTT 222

Installing Mosquitto MQTT Broker on a Raspberry Pi 227

MQTT – Establishing a Two-way Communication 231

Installing Node-RED and Node-RED Dashboard on a Raspberry Pi 246

MQTT - Connect ESP32/ESP8266 to Node-RED 253

MODULE 6

Other components: OLED, DC Motor, Addressable RGB LED Strip 274

OLED Display 275

Control a DC Motor 285

Control an Addressable RGB LED Strip 299

MODULE 7

Thonny IDE 316

Thonny IDE 317

Flashing MicroPython Firmware (esptool.py) 318

Installing Thonny IDE 333

Thonny IDE Overview.....345

Troubleshooting Tips for Thonny IDE 353

Appendix: List of Parts Required 358

Other RNT Courses/eBooks 360

MODULE 0

Course Intro

Welcome to MicroPython Programming with ESP32 and ESP8266

Welcome to "MicroPython Programming with ESP32 and ESP8266". This is a practical course where you'll learn how to program the ESP32 and ESP8266 using MicroPython firmware. MicroPython is a re-implementation of Python 3 programming language targeted for microcontrollers and embedded systems. The content provided throughout this eBook is compatible with both the ESP32 and ESP8266 boards.

How to Follow this Course?

If you've never programmed the ESP32/ESP8266 using MicroPython firmware, we recommend following the eBook in order. We start with the most basic and simple concepts, and then we'll introduce new concepts along the way.

If you already have some experience programming with MicroPython you can skip the most basic sections and go straight to the Modules you find more interesting.

Download Source Code and Resources

Each section contains the source code, schematics, and all the resources you need to follow the projects and tutorials. You can download each resource by clicking the "Source Code" boxes that appear on each project, or you can download the MicroPython Programming with ESP32/ESP8266 GitHub repository and instantly download all the resources for this course.

Getting Parts for the Course

To follow this course, you need some electronics components. In each section, we provide a complete list of the needed parts and links to Maker Advisor, so that you can find the part you're looking for on your favorite store at the best price.

If you buy your parts through Maker Advisor links, we'll earn a small affiliate commission (you won't pay more for it). By getting your parts through our affiliate links you are supporting our work. If there's a component or tool you're looking for, we advise you to take a look at our favorite tools and parts here.

Note: for the full parts list, consult the Appendix here.

Leave Feedback

Your feedback is very important so that we can improve the course and our learning materials. Suggestions, rectifications, and your opinion is very important for us.

You can use the following channels to leave feedback:

- Facebook group
- RNT Lab Forum

MODULE 1

Getting Started with MicroPython on ESP32 and ESP8266

Introducing MicroPython

In this section you'll get started with MicroPython firmware on the ESP32 and ESP8266. We'll introduce you to MicroPython, show you the differences between MicroPython and regular Python, and how to program your ESP based boards with MicroPython using uPyCraft IDE. After completing this section, you'll have your first LED blinking using MicroPython.

What is MicroPython?

MicroPython is a re-implementation of Python 3 programming language targeted for microcontrollers and embedded systems. MicroPython is very similar to regular Python. Therefore, if you already know how to program in Python, you also know how to program in MicroPython.

Python vs MicroPython

Apart from a few exceptions, the language features of Python are also available in MicroPython. The biggest difference between Python and MicroPython is that MicroPython was designed to work under constrained conditions.

Because of that, MicroPython does not come with the full standard library. It includes a small subset of the Python standard library. However, it does include modules to access low-level hardware – this means that there are libraries to easily access and interact with the GPIOs. Additionally, devices with Wi-Fi capabilities like the ESP32 and ESP8266 include modules to support network connections.

Why MicroPython?

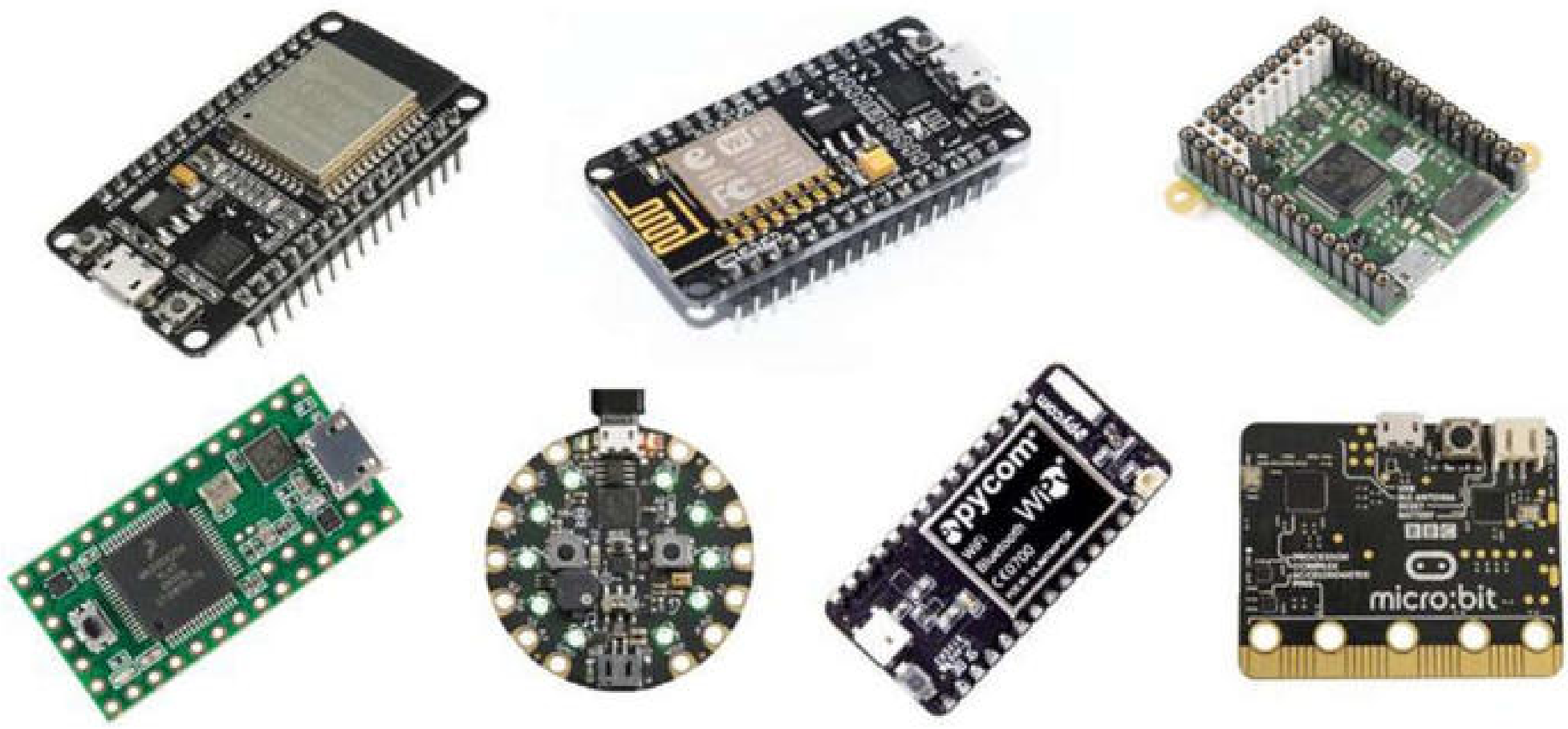
Python is one of the most widely used, simple and easy-to-learn programming languages around. So, the emergence of MicroPython makes it extremely easy and simple to program digital electronics. If you've never programmed digital electronics before, MicroPython is a good way to start.

MicroPython's goal is to make programming digital electronics as simple as possible, so that anyone can use it. Currently, MicroPython is used by hobbyists, researchers, teachers, educators, and even in commercial products. The code for blinking an LED on an ESP32 or ESP8266 is as simple as follows:

```
from machine import Pin
from time import sleep

led = Pin(2, Pin.OUT)

while True:
  led.value(not led.value())
  sleep(0.5)
```

MicroPython – Boards support

MicroPython runs on many different devices and boards, such as:



- ESP32
- ESP8266
- PyBoard
- Micro:Bit
- Teensy 3.X
- WiPy – Pycom
- Adafruit Circuit Playground Express
- Other ESP32/ESP8266 based boards

Click the following links for more information about other boards that support MicroPython:

- Boards running MicroPython – MicroPython Forum
- Boards summary – MicroPython GitHub

Throughout this eBook, we'll use MicroPython with the ESP32 and ESP8266 boards.

Note: if this is your first time with the ESP32 or ESP8266, we have a quick getting started guide for these boards after the MicroPython introduction.

ESP32 and ESP8266 boards are similar, and you won't feel almost any difference programming those using MicroPython. This means that anything you write for the ESP32 should also run with no changes or minimal changes on the ESP8266 (mainly changing the pin assignment).

Limitations

ESP32 is the successor of the ESP8266. So, currently, not all features are available in MicroPython to take the most out of the ESP32 – it's still an ongoing project. However, it's very usable and you can make all the presented projects with it.

uPyCraft IDE

There are different programs you can use to program the ESP32 and ESP8266 boards. To program the ESP32 or ESP8266 boards with MicroPython firmware, we recommend using **uPyCraft IDE** because it is simple, intuitive to use and works great with the ESP boards. However, if you are used to other MicroPython programming environment, you can stick with it.

uPyCraft IDE runs in any major operating system. We have instructions for installing uPyCraft IDE in Window, Mac OS X and Linux Ubuntu. Install uPyCraft IDE in your computer by following one the installation procedures we describe in the following Units.

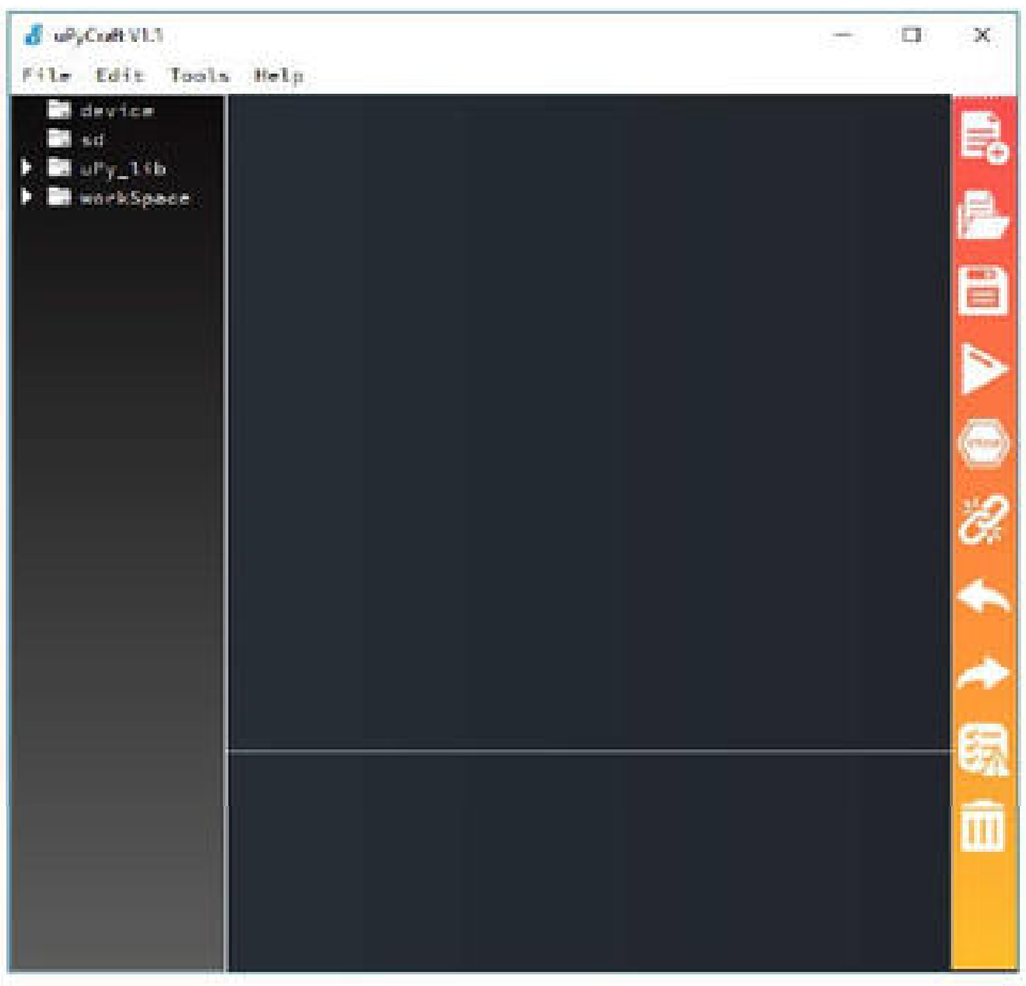
Alternative – Thonny IDE

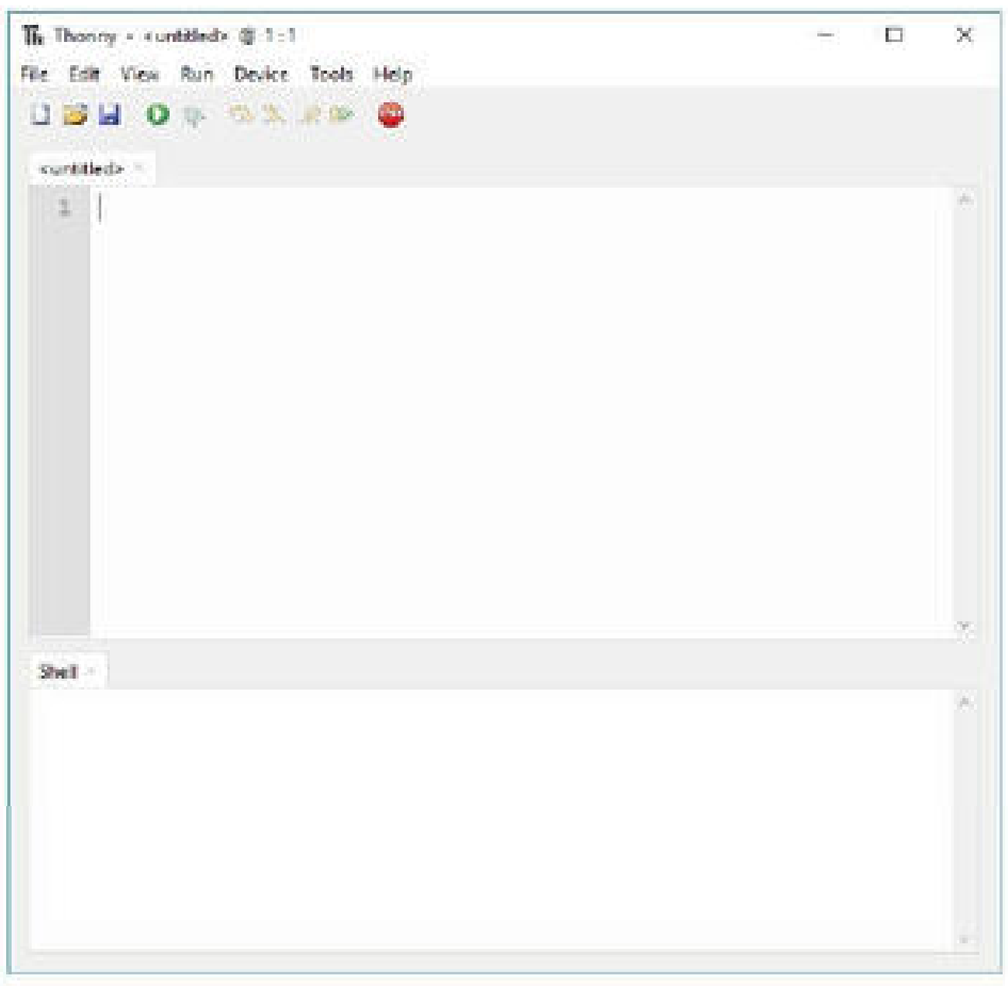
We've experimented with several IDEs to program the ESP32 and ESP8266 boards using MicroPython, and uPyCraft IDE worked fine for us. However, some users reported some trouble using uPyCraft IDE. So, we provide an alternative IDE to program your ESP32/ESP8266: **Thonny IDE**.

Note: we have a dedicated section to Thonny IDE that shows how to install and use it with the ESP32/ESP8266. Read "Getting Started with Thonny IDE".

The main differences between uPyCraft IDE and Thonny IDE:

- uPyCraft is an IDE for MicroPython, whereas Thonny is an IDE for Python with MicroPython support;
- uPyCraft IDE prints all the debug information in the debugging window which can be confusing. Thonny IDE has a layer on top that doesn't show all those errors, so it's a bit more user friendly;
- uPyCraft IDE has more features dedicated to program ESP8266/ESP32 with MicroPython, but it's harder to install on some Linux computers. Thonny IDE is easier to install and compatible with all operating systems;
- uPyCraft provides a better way to edit and delete files saved on your boards. On the other hand, Thonny IDE has a built-in feature to upload files as *boot.py* or *main.py*. That feature makes it easier and faster to upload new scripts.





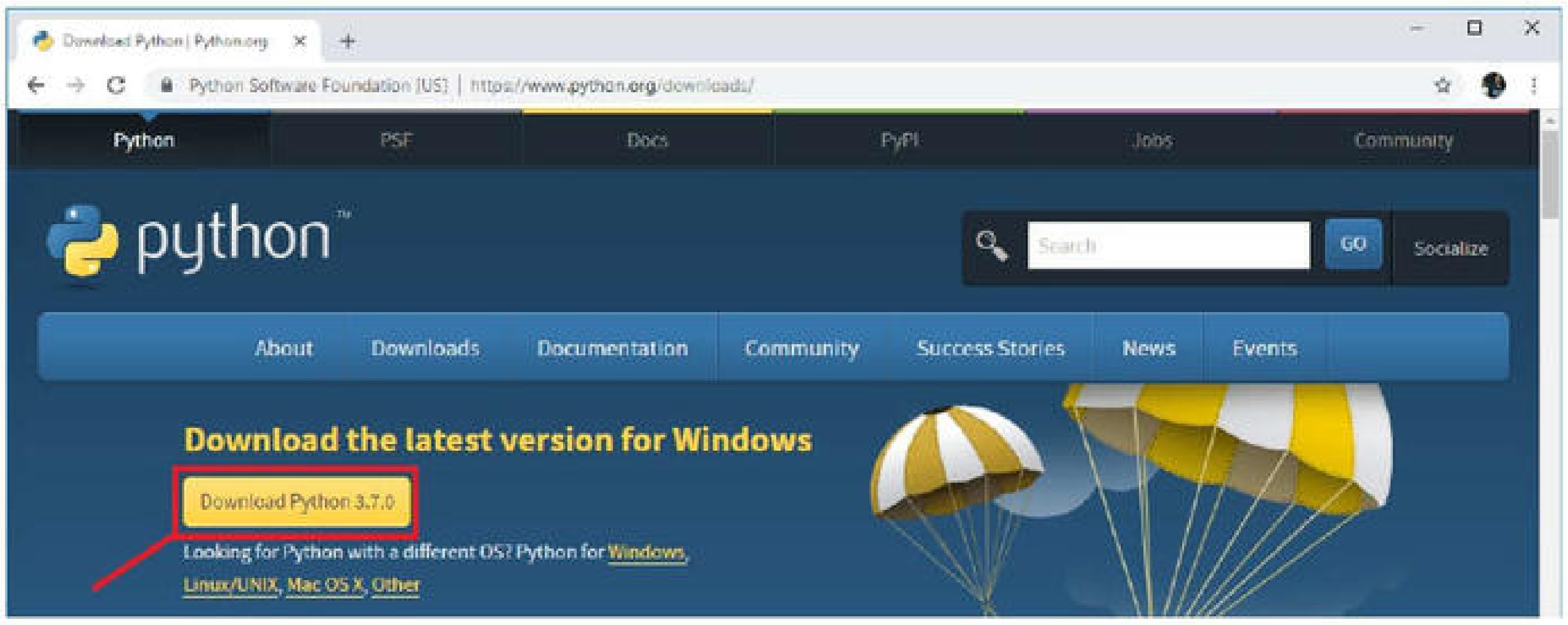
Installing uPyCraft IDE (Windows)

This section shows how to install uPyCraft IDE in your Windows PC. The IDE is a software that contains tools to make the process of development, debugging and upload code easier.

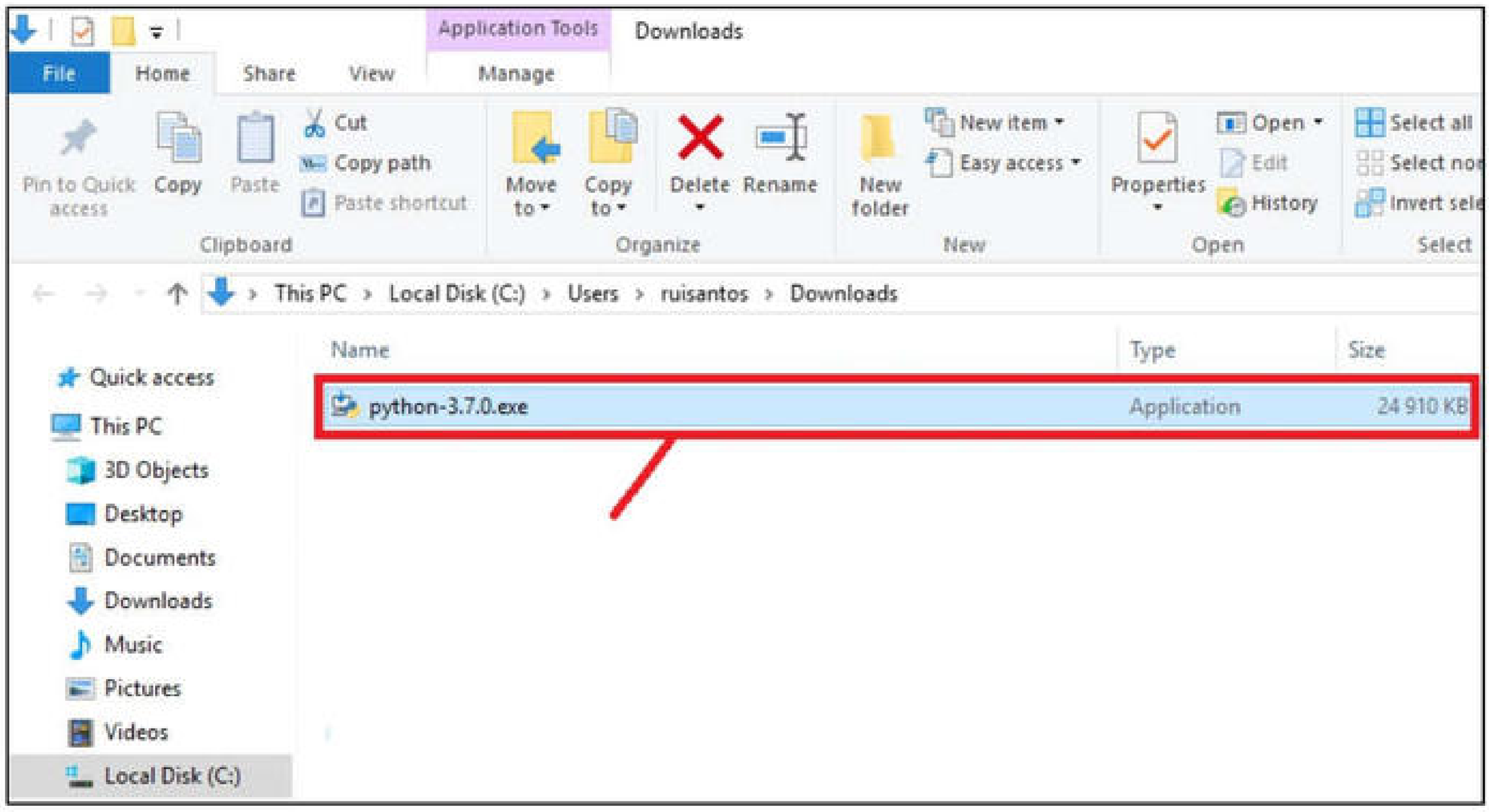
Installing Python 3.7.X

Before installing uPyCraft IDE, make sure you have the latest version of **Python 3.7.X** installed in your computer. Follow the next installation instructions:

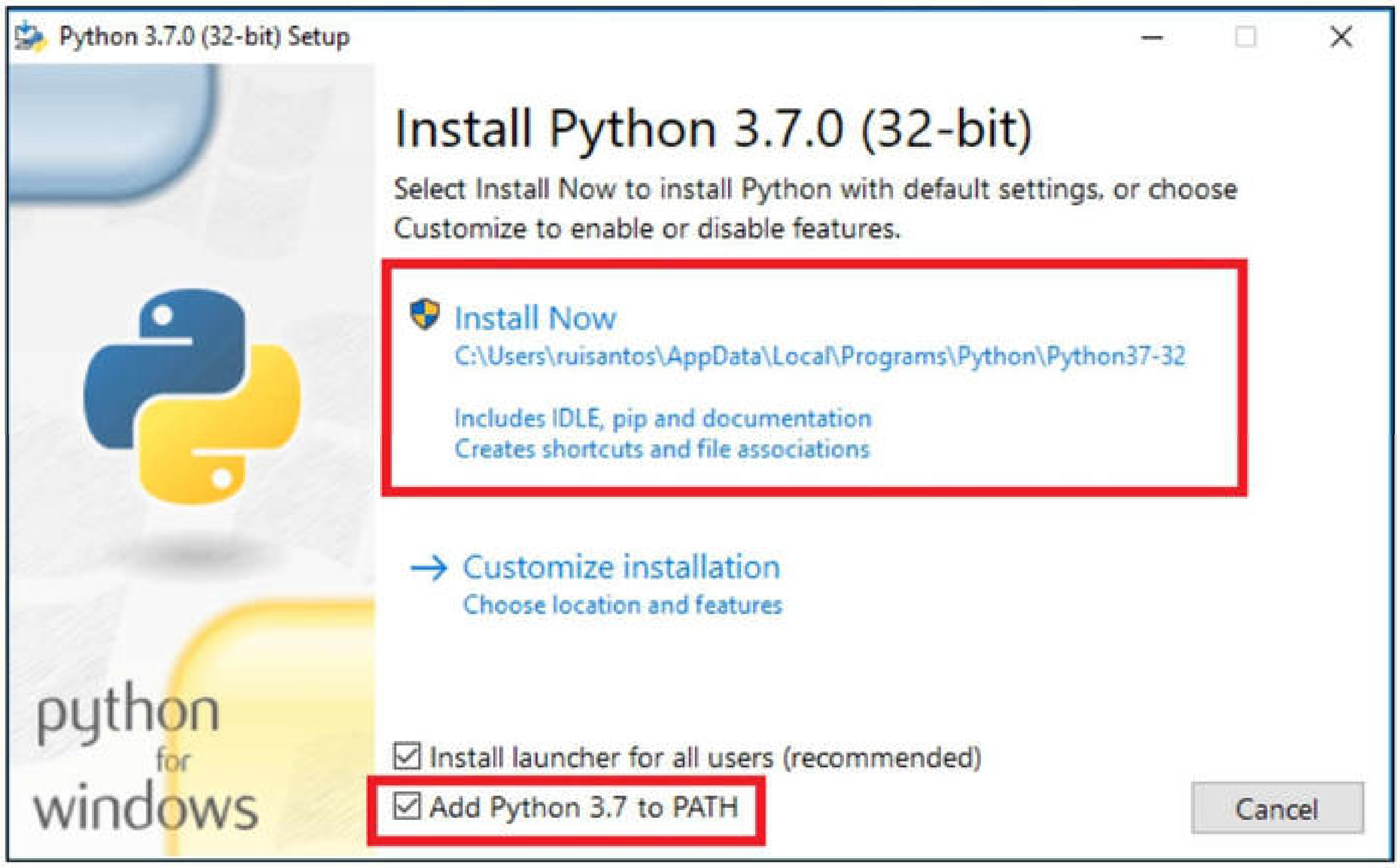
1. Go to the Python Downloads page: www.python.org/downloads and download the installation file.



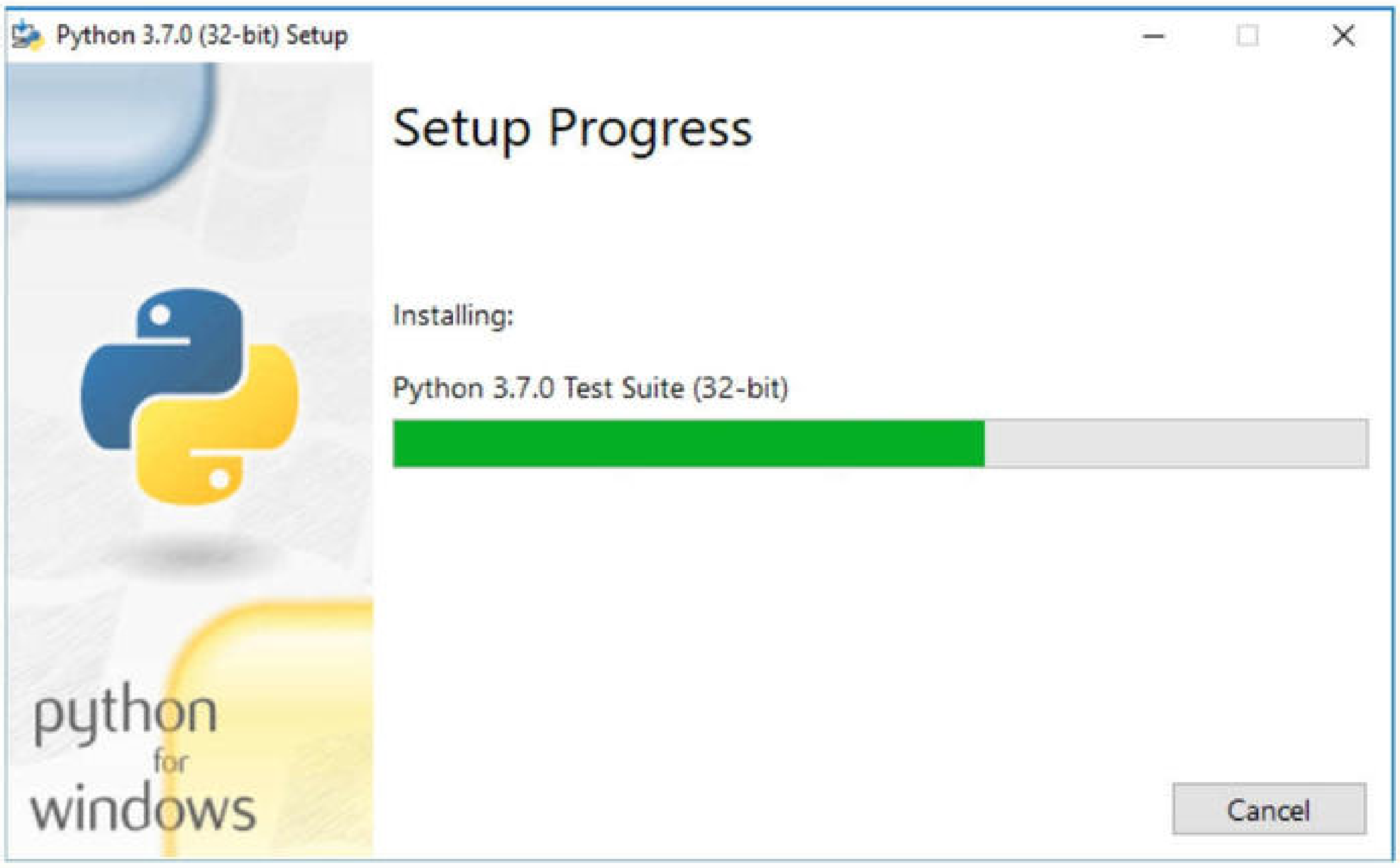
2. After a few seconds, you should have a file called python-3.7.X.exe file in your computer. Double-click the file to open it.



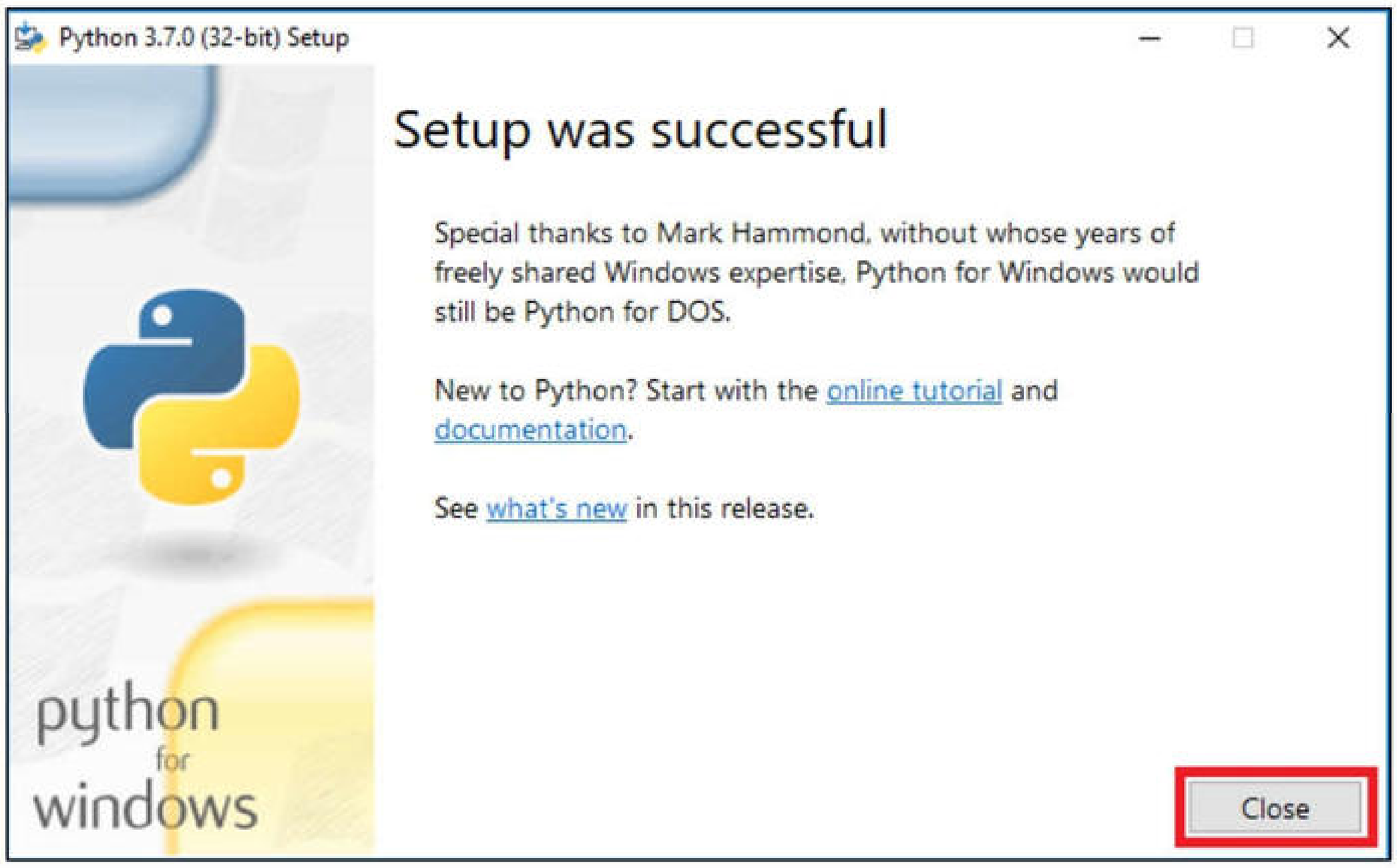
3. Enable the option at the bottom "**Add Python 3.7 to PATH**". Then, press the "**Install Now**" button:



4. Wait a few seconds while the software completes the installation process.



5. When it's done, you should see the message "**Setup was successful**" and you can close that window.



Installing uPyCraft IDE – Windows PC

As mentioned before, throughout this book we'll be using uPyCraft IDE to program the ESP32 or ESP8266 boards using the MicroPython firmware. In our opinion, right now uPyCraft IDE is the easiest way of programming ESP based boards with MicroPython.

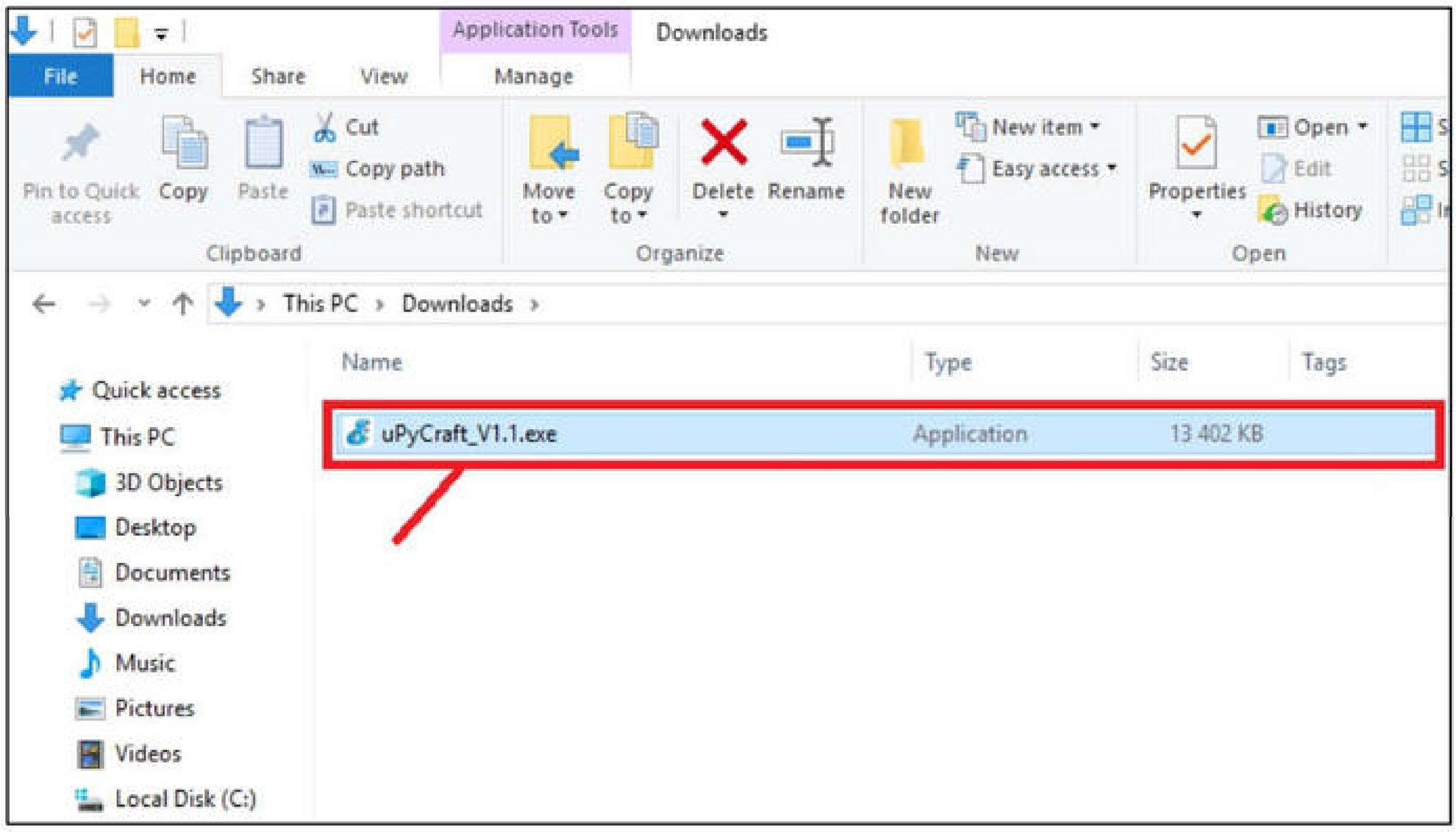
You can learn more about uPyCraft IDE on their GitHub repository or explore the uPyCraft IDE source code.

Downloading uPyCraft IDE for Windows

Click here to download uPyCraft IDE for Windows or go to the following link:

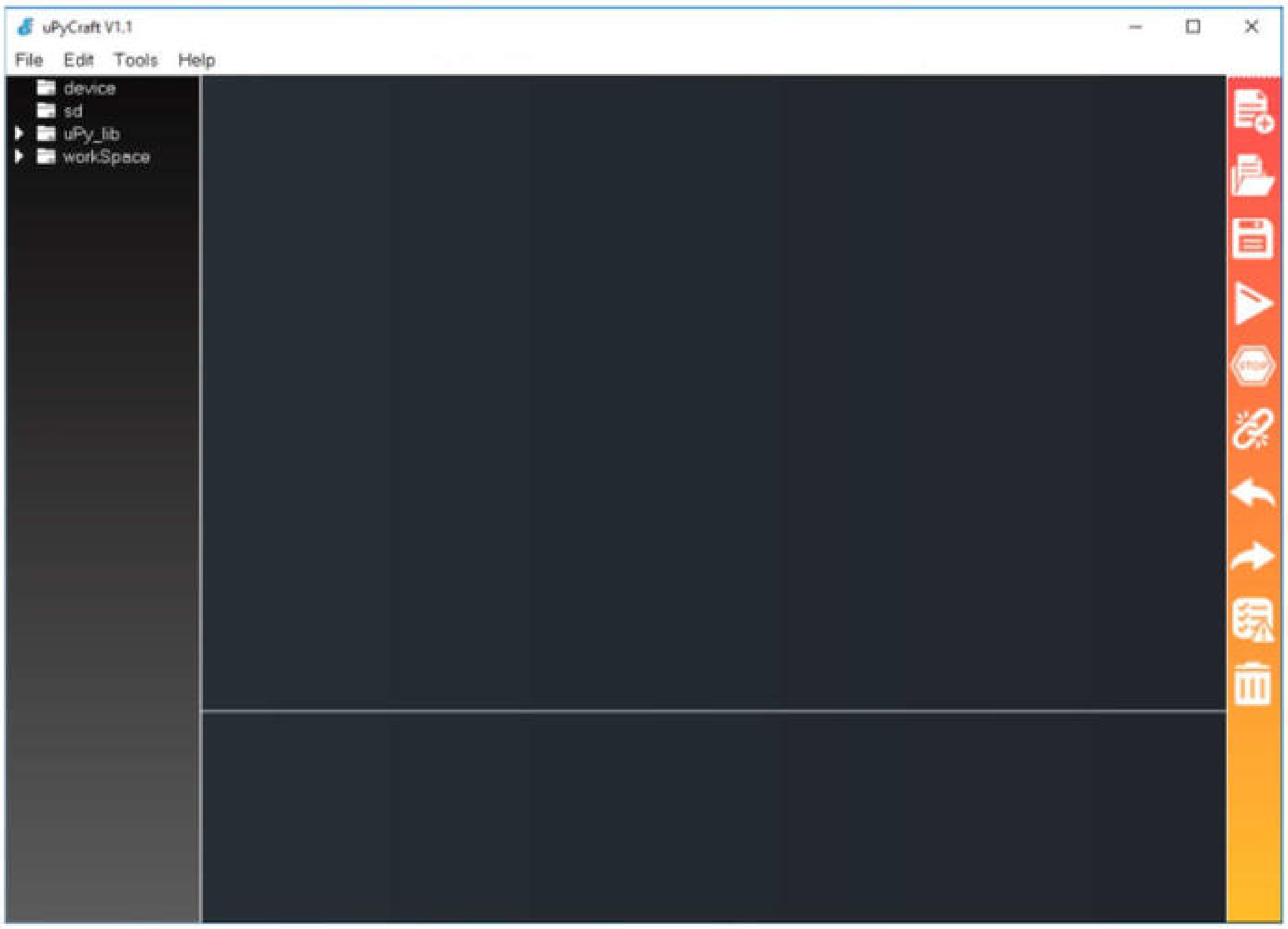
- https://randomnerdtutorials.com/uPyCraftWindows

After a few seconds you should see a similar file (**uPyCraft_VX.exe**) in your *Downloads* folder:



Note: uPyCraft IDE is an open-source project and all the source code is available online. Most antivirus won't allow you to download *.exe* files from the internet, and they will flag the uPyCraft file as suspicious. Make sure you tell your antivirus that uPyCraft IDE is not virus in order to be able to run it.

Double-click the downloaded file. A new window opens with the uPyCraft IDE software:



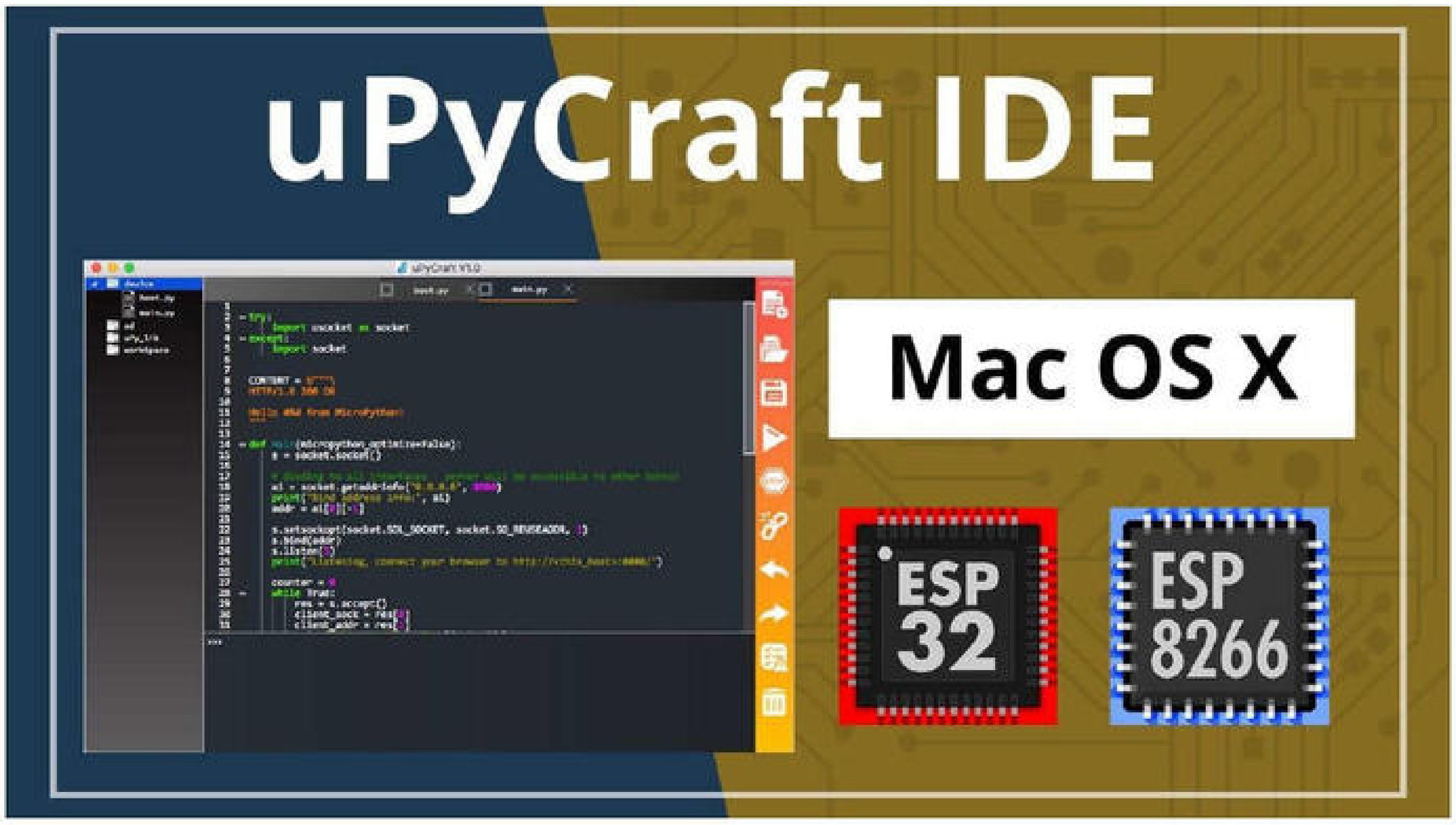
We'll be using this software to flash the ESP based boards with MicroPython firmware as well as to program the boards.

After installing the uPyCraft software, follow the "**Flashing MicroPython Firmware to ESP32/ESP8266**" Unit:

- Flash MicroPython – ESP32
- Flash MicroPython – ESP8266

Note: if you have trouble using uPyCraft IDE, you can try using Thonny IDE instead. Just follow the Thonny IDE section.

Install uPyCraft IDE (Mac OS X)



This section shows how to install uPyCraft IDE in your Mac OS X computer. The IDE is a software that contains tools to make the process of development, debugging and upload code easier.

Installing Python 3.X – Mac OS X

Before installing the uPyCraft IDE, make sure you have the latest version of **Python 3.X** installed in your computer. If you don't, follow the next instructions to install Python 3.X with the brew command. Open a terminal window and type the next command:

```
$ brew install python3
```

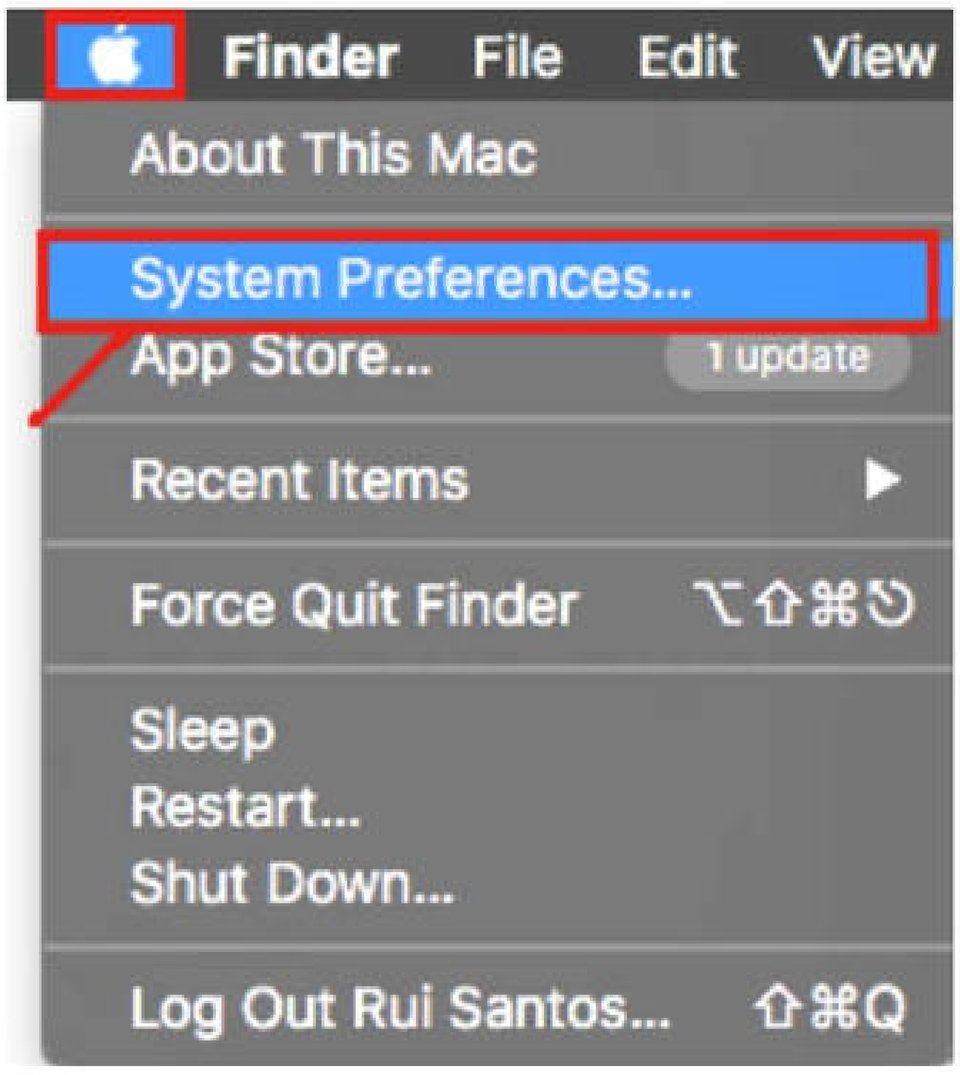
Wait a few seconds while the installation procedure is completed.

```
Rui — -bash — 75×19
Last login: Tue Oct  2 23:10:50 on console
Utilizadors-MacBook-Pro:~ Rui$ brew install python3
Updating Homebrew...
==> Auto-updated Homebrew!
Updated 1 tap (homebrew/core).
==> New Formulae
bundletool                 mallet                     stanford-corenlp
geant4                     opentracing-cpp            tdlib
==> Updated Formulae
git ✔                 exiftool              libssh                serverless
aircrack-ng           faas-cli              nginx                 skafos
ammonite-repl         fpc                   node-build            smlnj
ansible               glances               openapi-generator     swiftformat
aws-okta              go                    opencv                swiftgen
babel                 godep                 pandoc                terragrunt
carthage              golang-migrate        pegtl                 topgrade
cern-ndiff            grv                   pixz                  unrar
cmocka                haproxy               presto                vice
commandbox            ipython               pulumi                vim
```

Allowing Apps Downloaded to Run

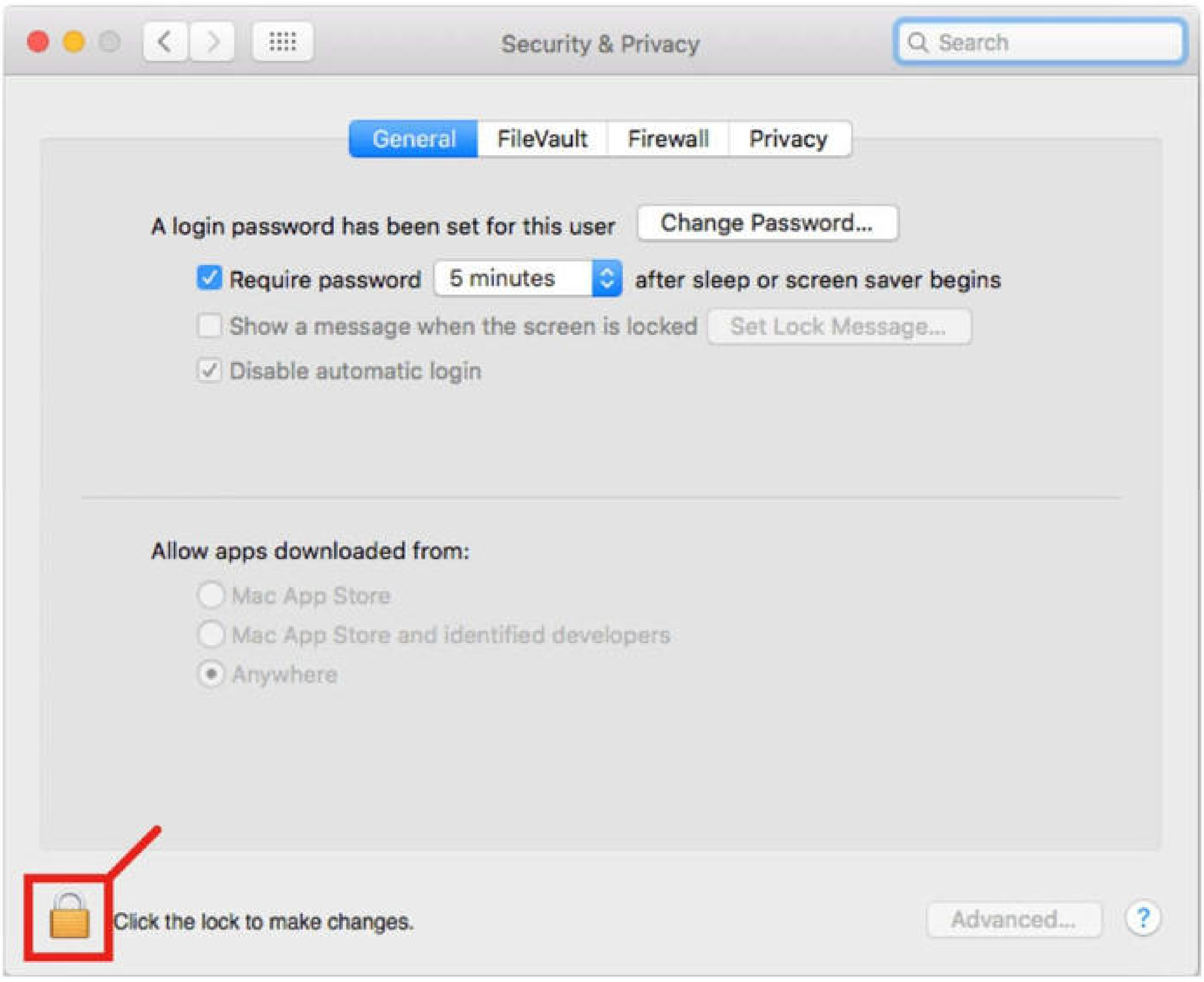
Since uPyCraft IDE is open source and downloaded from the Internet, it's not a verified app in the App Store. For security reasons, Mac OS X blocks unknown apps to run on your computer. Follow these next instructions to enable any downloaded software to run in your Mac.

1. Open the "**System Preferences...**" menu.



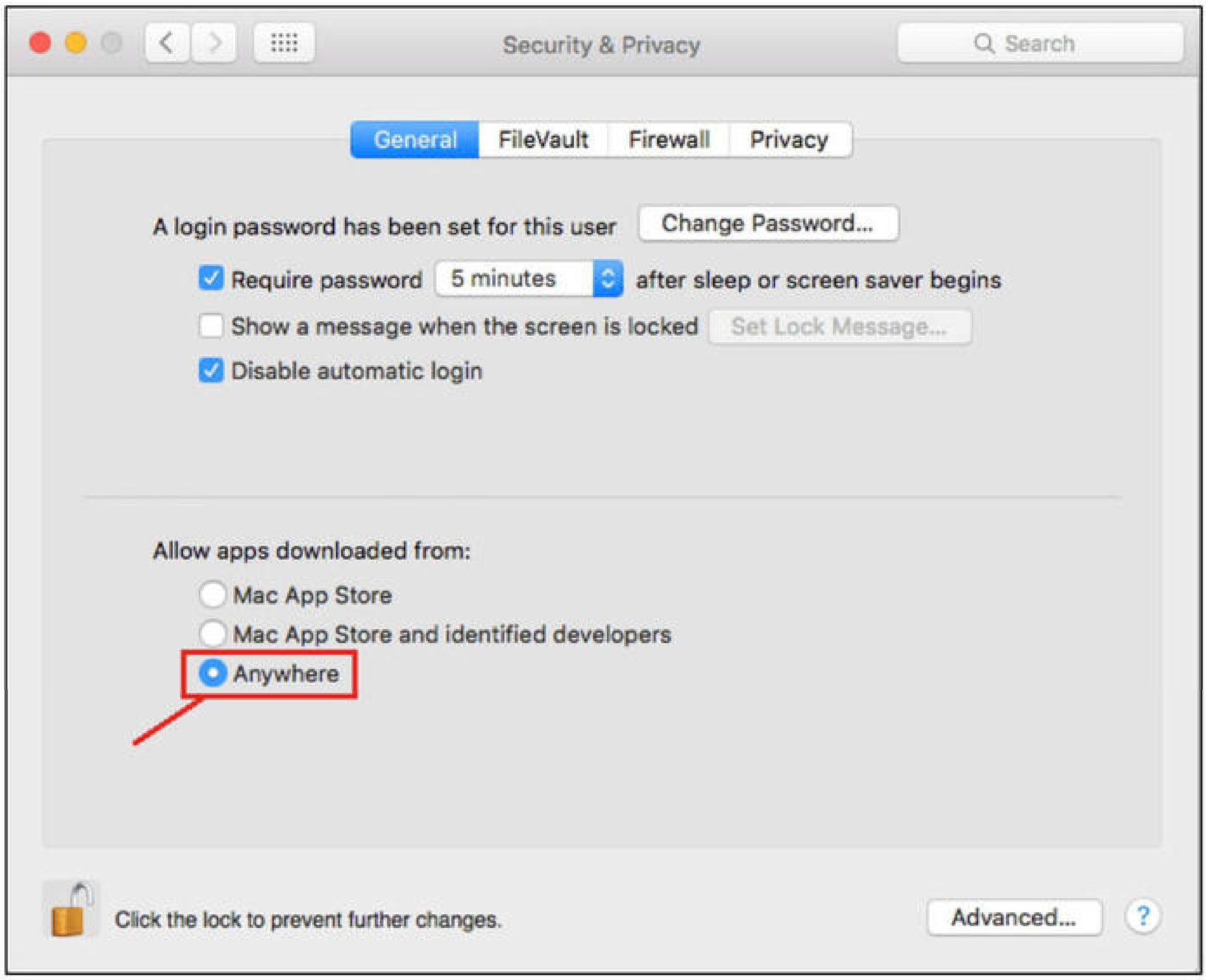
2. Open the "**Security & Privacy**" menu.

3. At the bottom left corner, click the lock icon to modify your "**Security & Privacy**" settings:



4. Type your username/password and click the "**Unlock**" button.

5. Finally, select the option "Allow apps downloaded from: **Anywhere**".



That's it, you can close that window.

Installing uPyCraft IDE – Mac OS X

As mentioned before, for this tutorial we'll be using uPyCraft IDE to program the ESP32 or ESP8266 boards using the MicroPython firmware. In our opinion, right now uPyCraft IDE is the easiest way of programming ESP based boards with MicroPython.

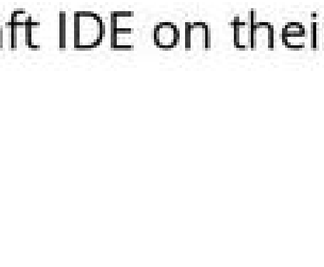
You can learn more about uPyCraft IDE on their GitHub repository or explore the uPyCraft IDE source code.

Downloading uPyCraft IDE for Mac OS X

Click here to download uPyCraft IDE for Mac OS X or go to the following link:

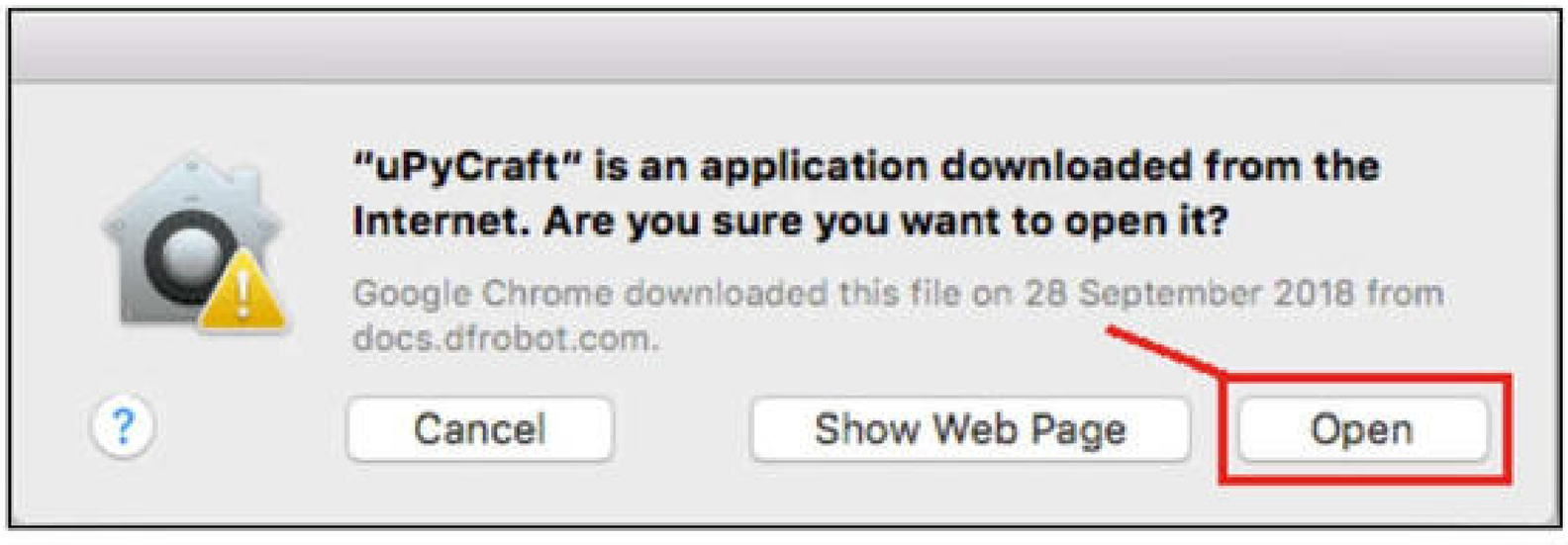
- https://randomnerdtutorials.com/uPyCraftMac

After a few seconds, you should see a similar file (**uPyCraft_mac_VX.zip**) in your **Downloads** folder:

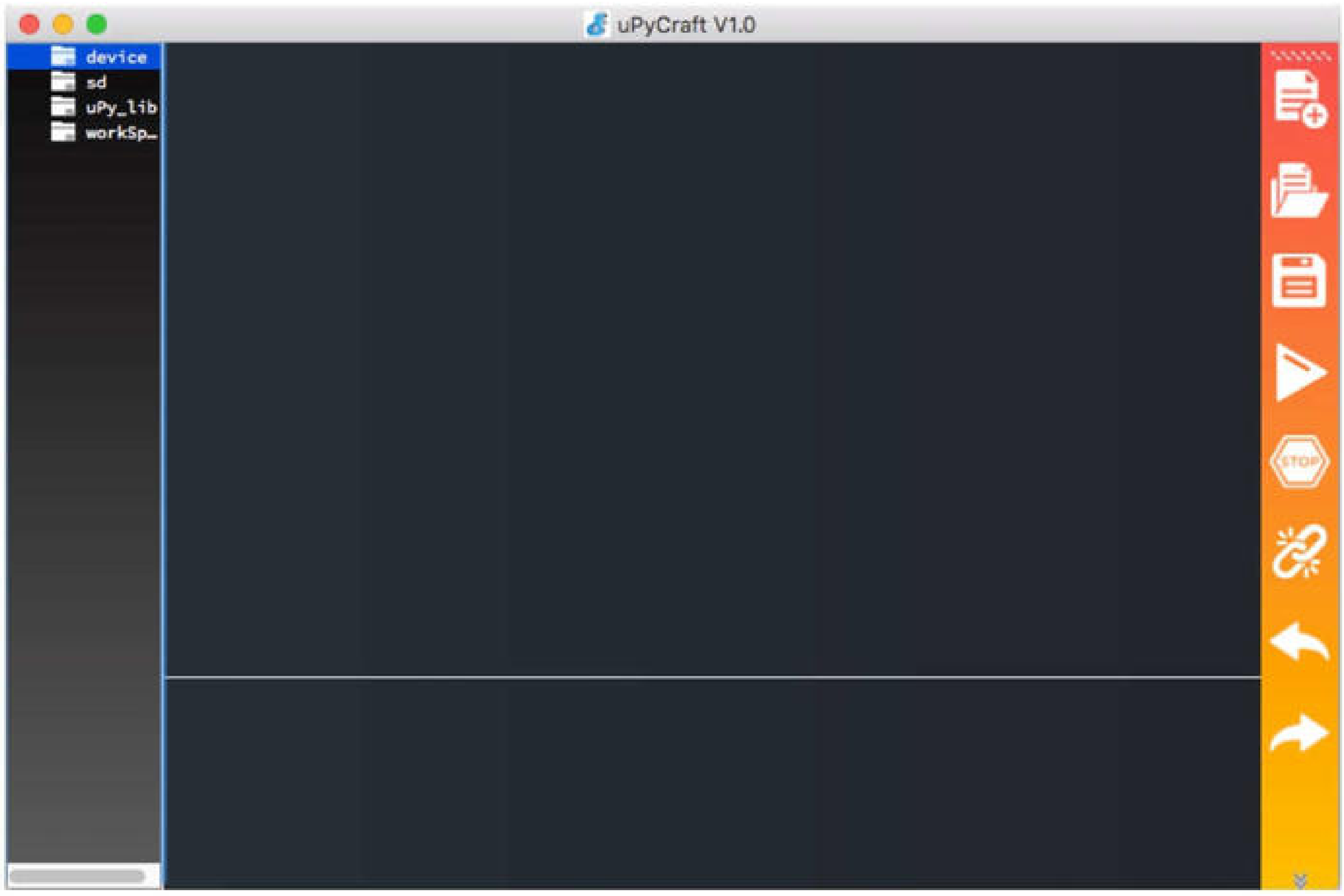


Unzip the **uPyCraft_mac_VX.zip** and you should see a new file called **uPyCraft**. Double-click the **uPyCraft** file:

Press the "**Open**" button to run it:



Here's what should open:



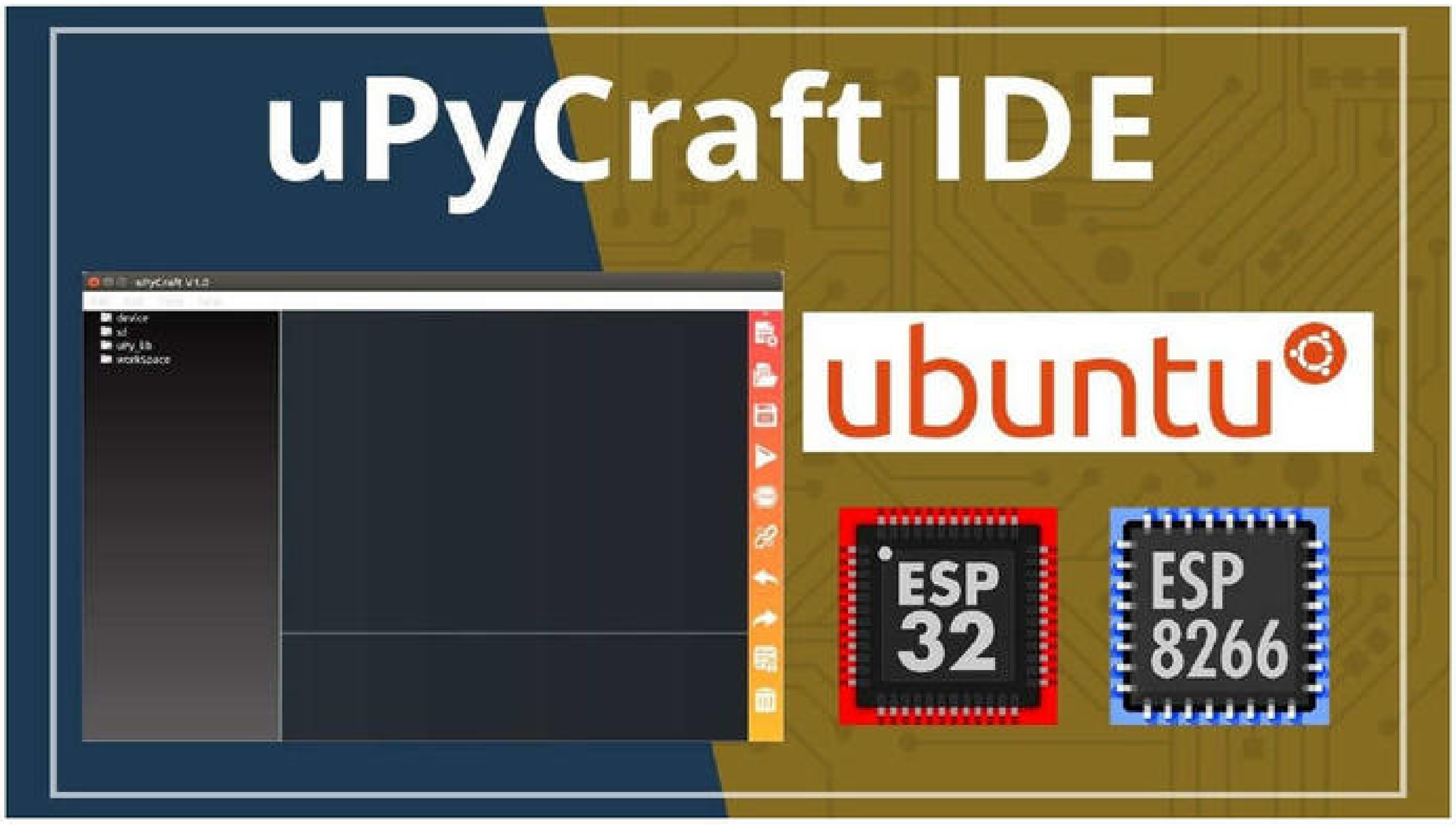
We'll be using this software to flash our ESP based boards with MicroPython firmware as well as to program the boards.

After installing the uPyCraft software, follow the "**Flashing MicroPython Firmware to ESP32/ESP8266**" Unit:

- Flash MicroPython – ESP32
- Flash MicroPython – ESP8266

Note: if you have trouble using uPyCraft IDE, you can try using Thonny IDE instead. Just follow the Thonny IDE section.

Install uPyCraft IDE (Linux Ubuntu)



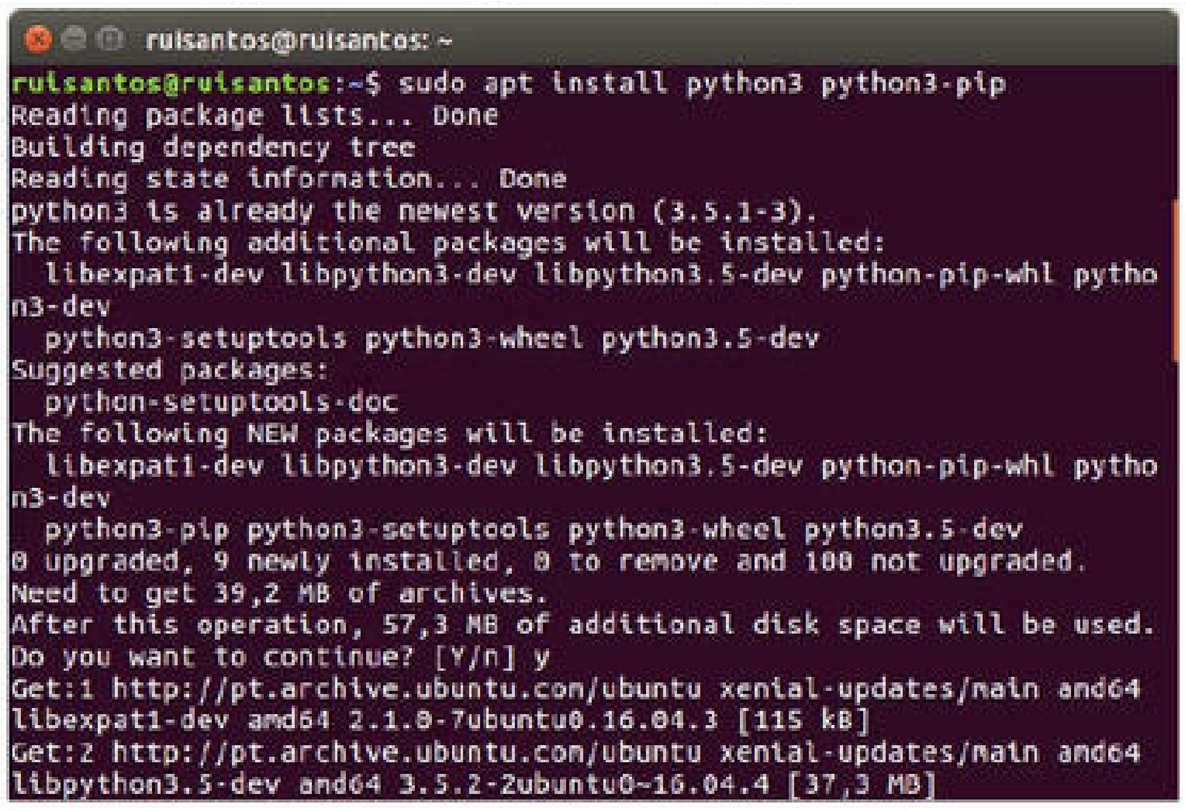
This section shows how to install uPyCraft IDE in Linux Ubuntu. The IDE is a software that contains tools to make the process of development, debugging and upload code easier.

Installing Python 3.X – Linux Ubuntu

Before installing the uPyCraft IDE, make sure you have **Python 3.X** installed in your computer. If you don't, follow the next instructions to install Python 3.X.

Run this command to install Python 3 and pip:

```
$ sudo apt install python3 python3-pip
```



```
ruisantos@ruisantos: ~
ruisantos@ruisantos:~$ sudo apt install python3 python3-pip
Reading package lists... Done
Building dependency tree
Reading state information... Done
python3 is already the newest version (3.5.1-3).
The following additional packages will be installed:
  libexpat1-dev libpython3-dev libpython3.5-dev python-pip-whl pytho
n3-dev
  python3-setuptools python3-wheel python3.5-dev
Suggested packages:
  python-setuptools-doc
The following NEW packages will be installed:
  libexpat1-dev libpython3-dev libpython3.5-dev python-pip-whl pytho
n3-dev
  python3-pip python3-setuptools python3-wheel python3.5-dev
0 upgraded, 9 newly installed, 0 to remove and 100 not upgraded.
Need to get 39,2 MB of archives.
After this operation, 57,3 MB of additional disk space will be used.
Do you want to continue? [Y/n] y
Get:1 http://pt.archive.ubuntu.com/ubuntu xenial-updates/main amd64
libexpat1-dev amd64 2.1.0-7ubuntu0.16.04.3 [115 kB]
Get:2 http://pt.archive.ubuntu.com/ubuntu xenial-updates/main amd64
libpython3.5-dev amd64 3.5.2-2ubuntu0~16.04.4 [37,3 MB]
```

Installing uPyCraft IDE – Linux Ubuntu 16.04

As mentioned before, for this tutorial we'll be using uPyCraft IDE to program the ESP32 or ESP8266 boards using the MicroPython firmware. In our opinion, right now uPyCraft IDE is the easiest way of programming ESP based boards with MicroPython.

You can learn more about uPyCraft IDE on their GitHub repository or explore the uPyCraft IDE source code.

IMPORTANT: at the time of writing this installation procedure, uPyCraft IDE is only tested on Linux Ubuntu 16.04. If you want to run it on a different Ubuntu version or Linux distribution, we recommend using uPyCraft IDE source code and compile the software yourself.

Downloading uPyCraft IDE for Linux Ubuntu 16.04

Click here to download uPyCraft IDE for Linux Ubuntu 16.04 or go to the following link:

- https://randomnerdtutorials.com/uPyCraftLinux

Open your Terminal window, navigate to your **Downloads** folder and list all the files:

```
$ cd Downloads
$ ls -l
  uPyCraft_linux_V1.X
```

You should have a similar file (**uPyCraft_linux_V1.X**) in your **Downloads** folder. You need to make that file executable with the following command:

```
$ chmod +x uPyCraft_linux_V1.X
```

Then, to open/run the uPyCraft IDE software type the next command:

```
$ ./uPyCraft_linux_V1.X
```

```
ruisantos@ruisantos: ~/Downloads
ruisantos@ruisantos:~$ cd Downloads/
ruisantos@ruisantos:~/Downloads$ ls -l
total 26620
-rw-rw-r-- 1 ruisantos ruisantos 27256160 Out  5 17:27 uPyCraft_linux_V1.0
ruisantos@ruisantos:~/Downloads$ chmod +x uPyCraft_linux_V1.0
ruisantos@ruisantos:~/Downloads$ ls -l
total 26620
-rwxrwxr-x 1 ruisantos ruisantos 27256160 Out  5 17:27 uPyCraft_linux_V1.0
ruisantos@ruisantos:~/Downloads$ ./uPyCraft_linux_V1.0
/home/ruisantos
/home/ruisantos/AppData/Local/uPyCraft/temp/
/home/ruisantos/AppData/Local/uPyCraft/examples
```

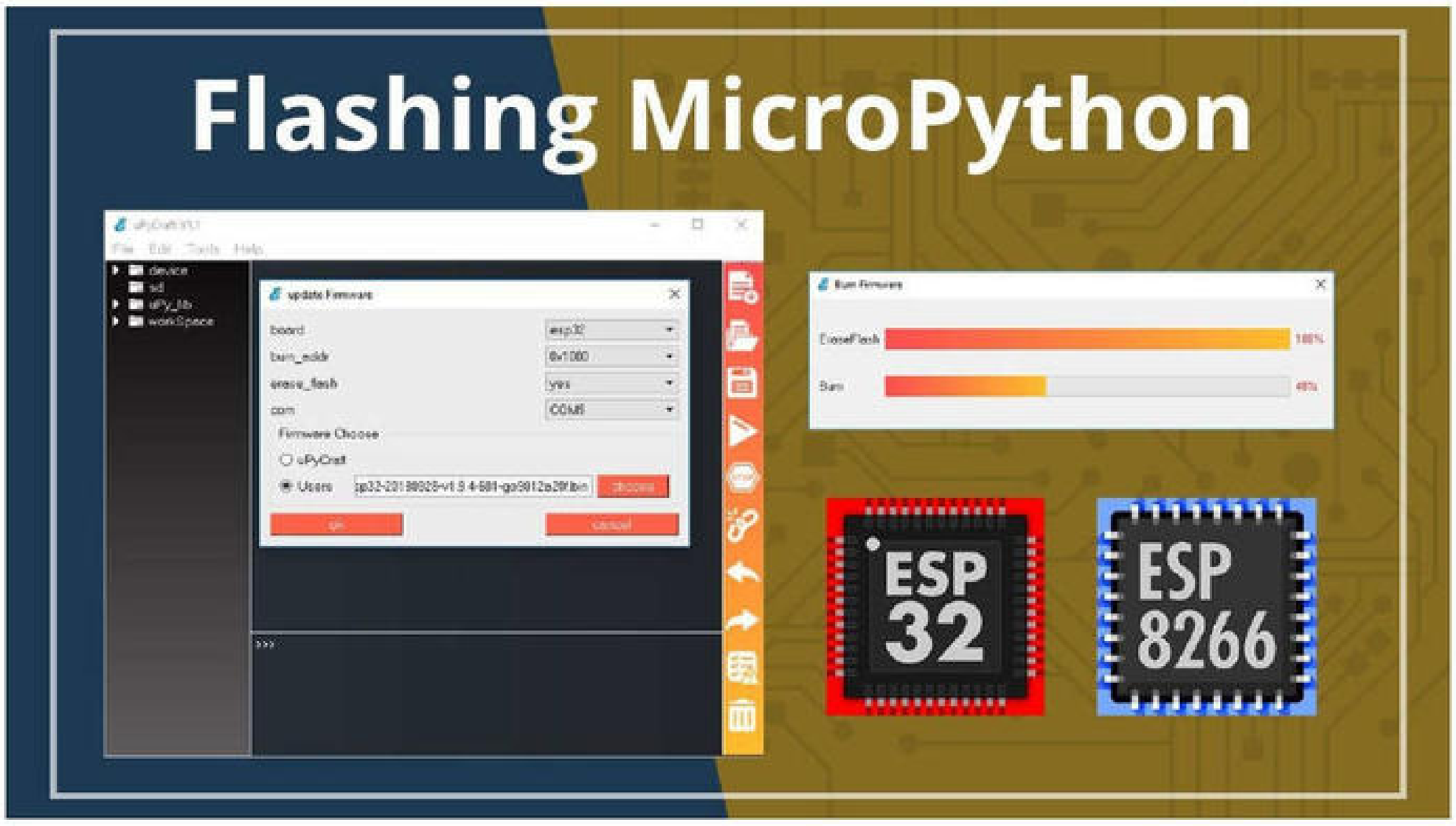
We'll be using the uPyCraft IDE software to flash our ESP based boards with MicroPython firmware as well as to program the boards.

After installing the uPyCraft software, follow the "**Flashing MicroPython Firmware to ESP32/ESP8266**" Unit:

- Flash MicroPython – ESP32
- Flash MicroPython – ESP8266

Note: if you have trouble using uPyCraft IDE, you can try using Thonny IDE instead. Just follow the Thonny IDE section.

Flashing MicroPython Firmware to ESP32/ESP8266



Unlike other boards, MicroPython isn't flashed onto the ESP32 or ESP8266 by default. That is the first thing you need to do to start programming your boards with MicroPython: flash/upload the firmware. Follow the next instructions to flash MicroPython firmware on your board.

Note: flashing your boards with MicroPython is reversible. This means that after you flash the ESP32/ESP8266 with MicroPython, you can still use Arduino IDE in the future. You just need to upload a code using the Arduino IDE to your board.

With uPyCraft IDE installed in your computer, you can easily flash your ESP32 or ESP8266 boards with the MicroPython firmware. Follow Part 1 or Part 2 depending on your board:

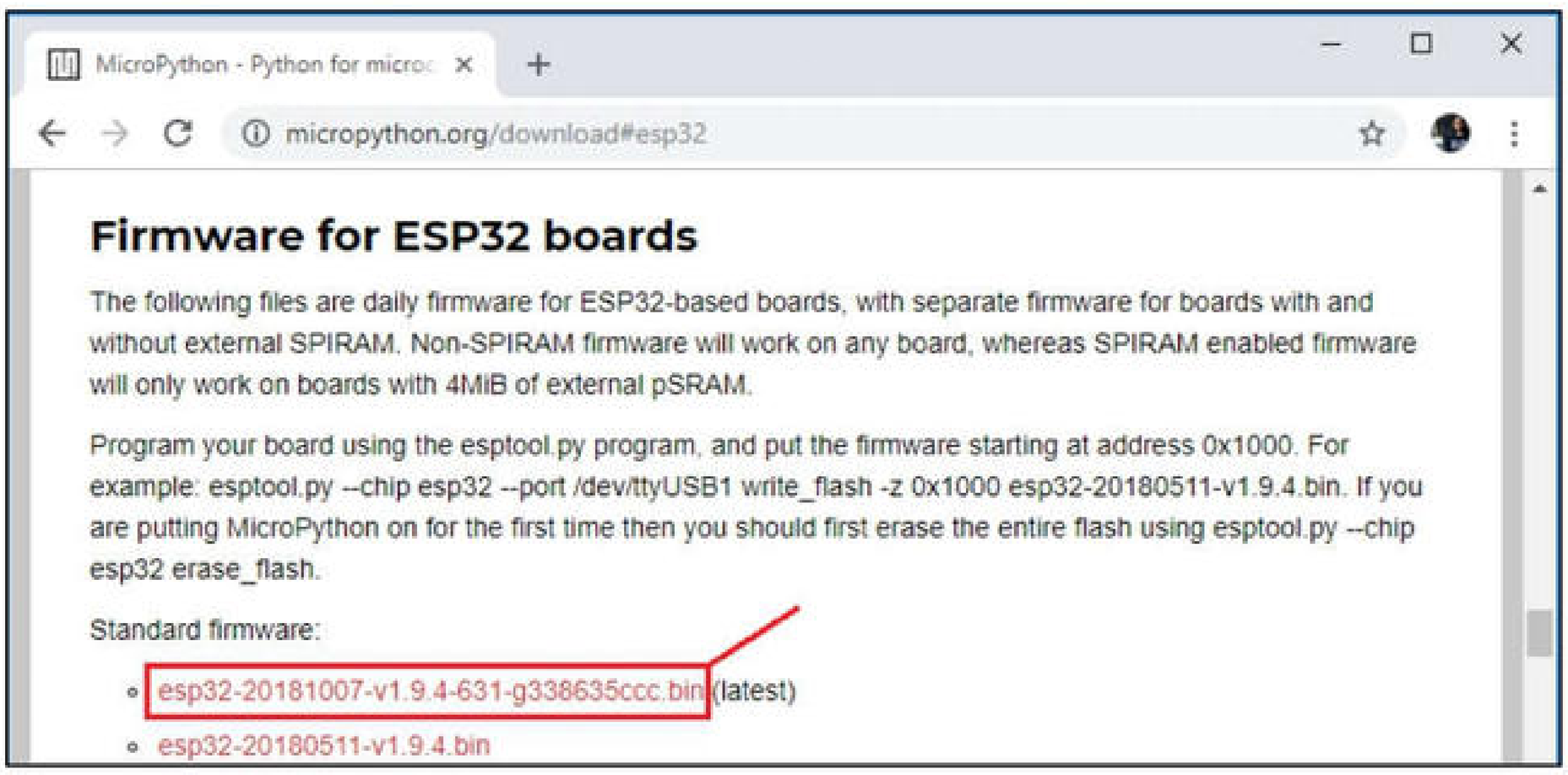
- Part 1 – ESP32
- Part 2 – ESP8266

[Part 1 – ESP32] Downloading and Flashing the MicroPython Firmware on ESP32

To download the latest version of MicroPython firmware for the ESP32, go to the MicroPython Downloads page and scroll all the way down to the ESP32 section.

You should see a similar web page (see figure below) with the latest link to download the ESP32 *.bin* file – for example:

esp32-20181007-v1.9.4-631-g338635ccc.bin.



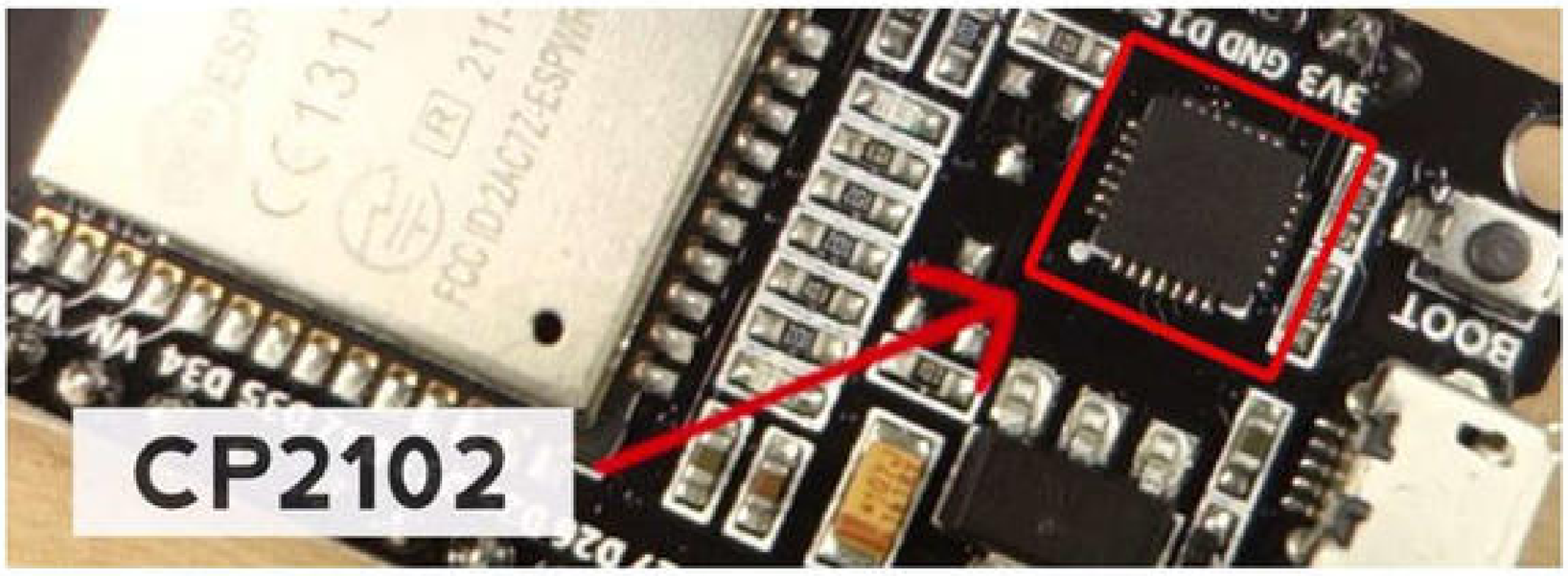
Selecting Serial Port

Connect your ESP32 board to your computer. Go to **Tools** ▸ **Serial** and select your ESP32 COM port (in our case it's COM5).

IMPORTANT: if you plug your ESP32 board to your computer, but you can't find the ESP32 Port available in your uPyCraft IDE, it might be one of these two problems: **1.** USB drivers missing or **2.** USB cable without data wires.

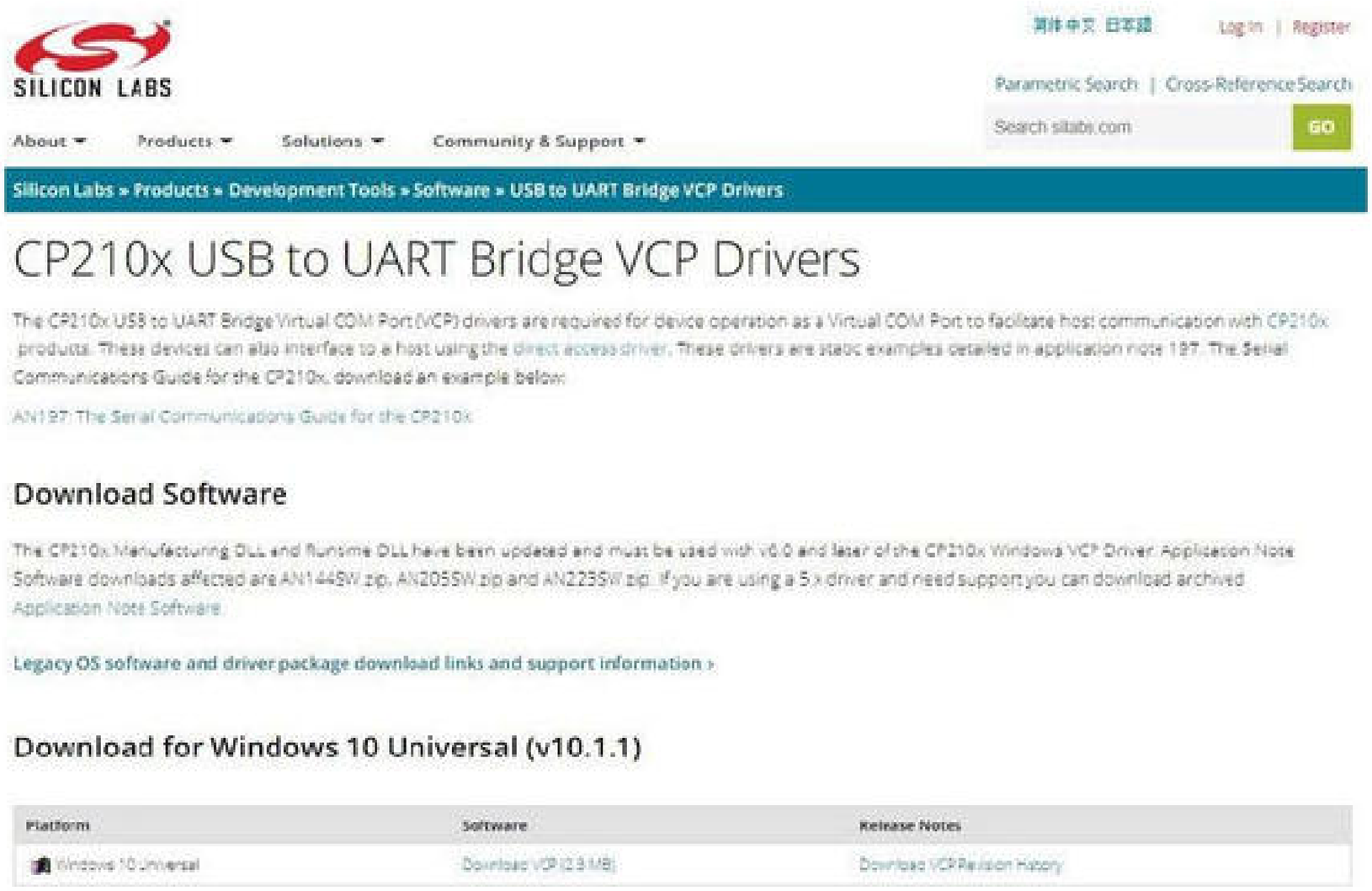
1. If you don't see your ESP's COM port available, this often means you don't have the USB drivers installed. Take a closer look at the chip next to the voltage regulator on board and check its name.

The ESP32 DEVKIT V1 DOIT board uses the CP2102 chip.



Go to Google and search for your specific chip to find the drivers and install them in your operating system.

You can download the CP2102 drivers on the Silicon Labs website.



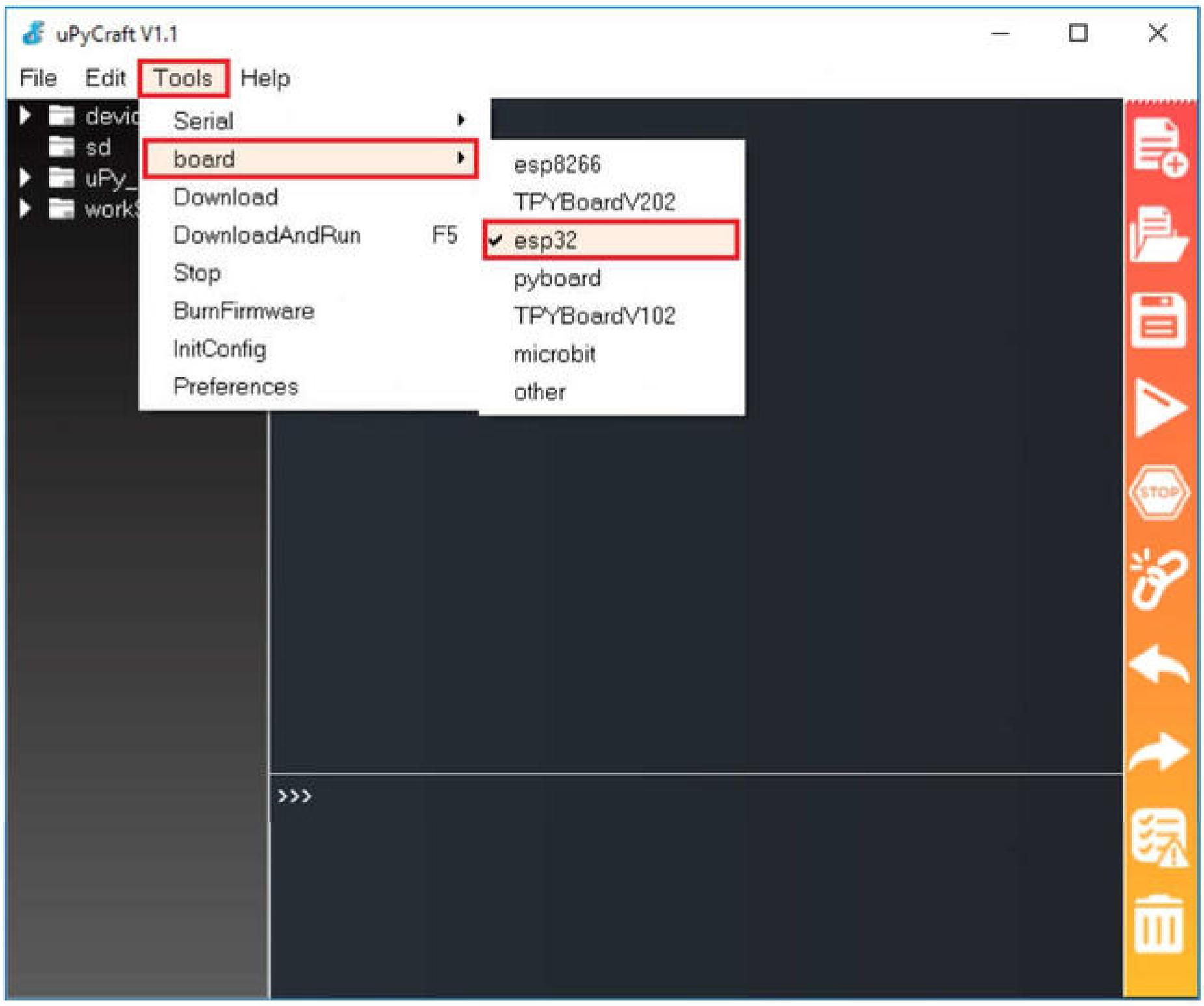
After they are installed, restart the uPyCraft IDE and you should see the COM port in the **Tools** menu.

2. If you have the drivers installed, but you can't see your device, double-check that you're using a USB cable with data wires.

USB cables from power banks often don't have data wires (they are charge only). So, your computer will never establish a serial communication with your ESP32. Using a proper USB cable should solve your problem.

Selecting the Board

Go to **Tools ▸ Board**. In this section, we assume that you're using the ESP32, so make sure you select the "**esp32**" option:

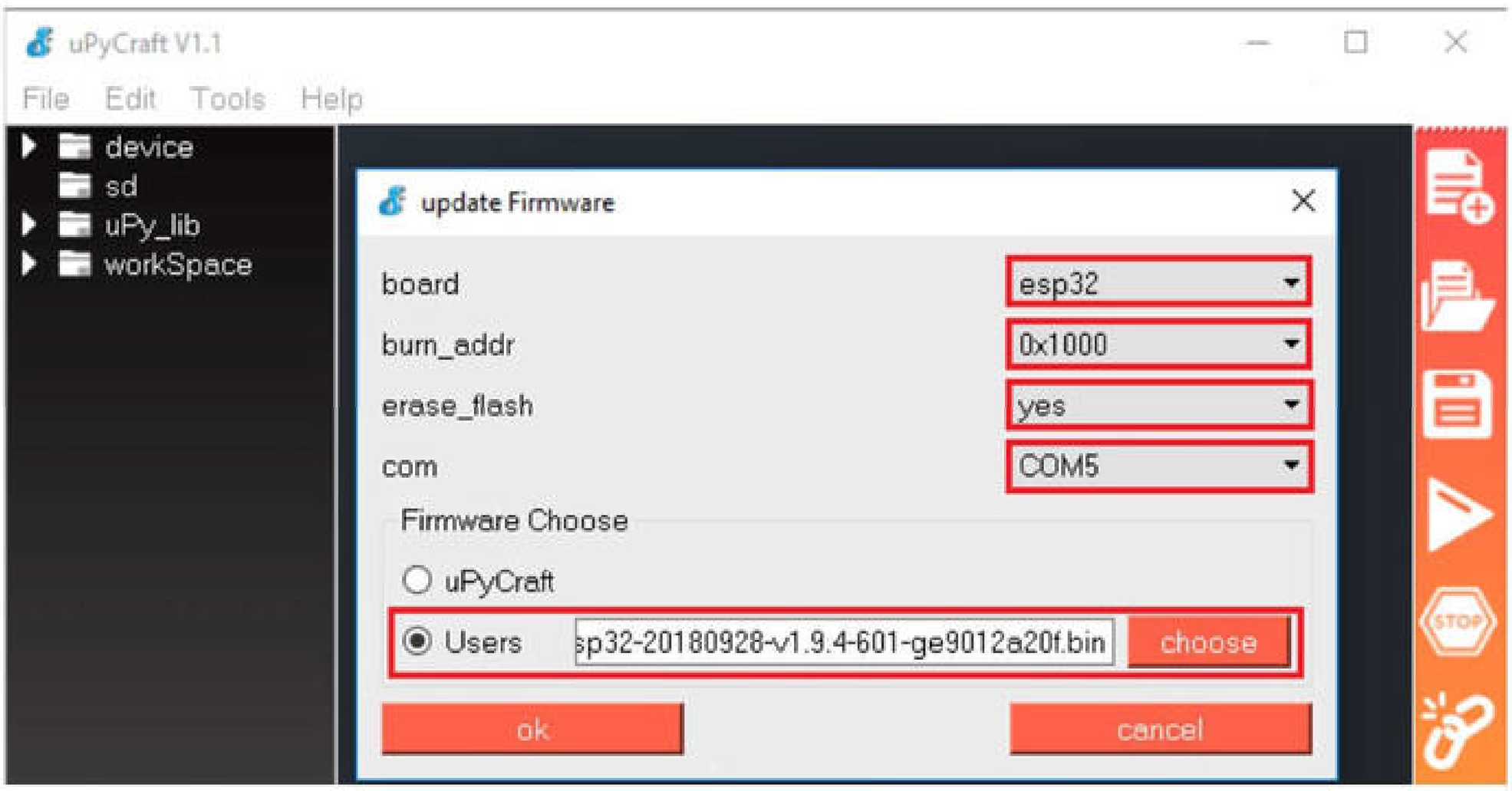


Flashing/Uploading MicroPython Firmware

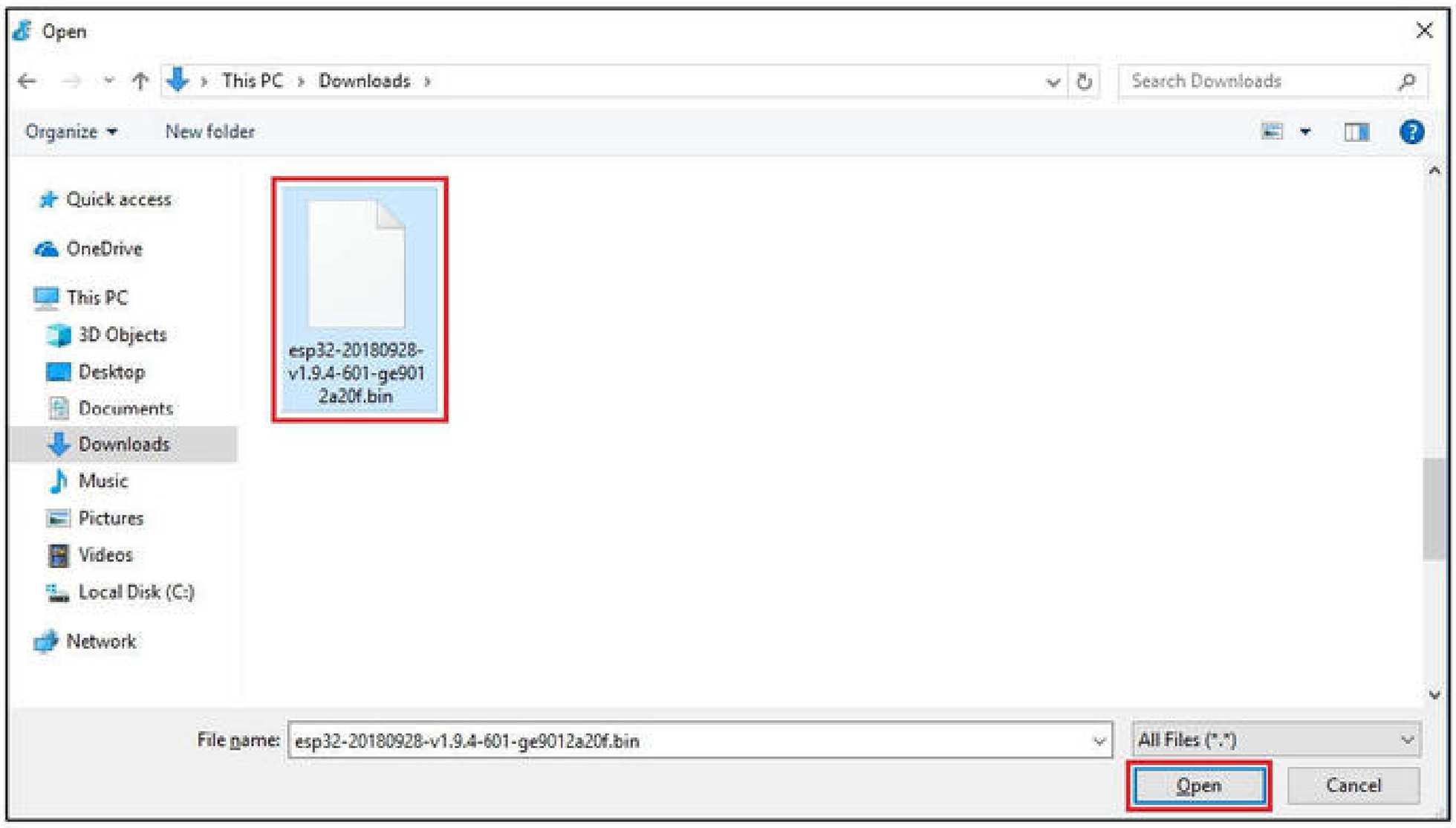
Finally, go to **Tools** ▸ **BurnFirmware** menu to flash your ESP32 with MicroPython.

Select all these options to flash the ESP32 board:

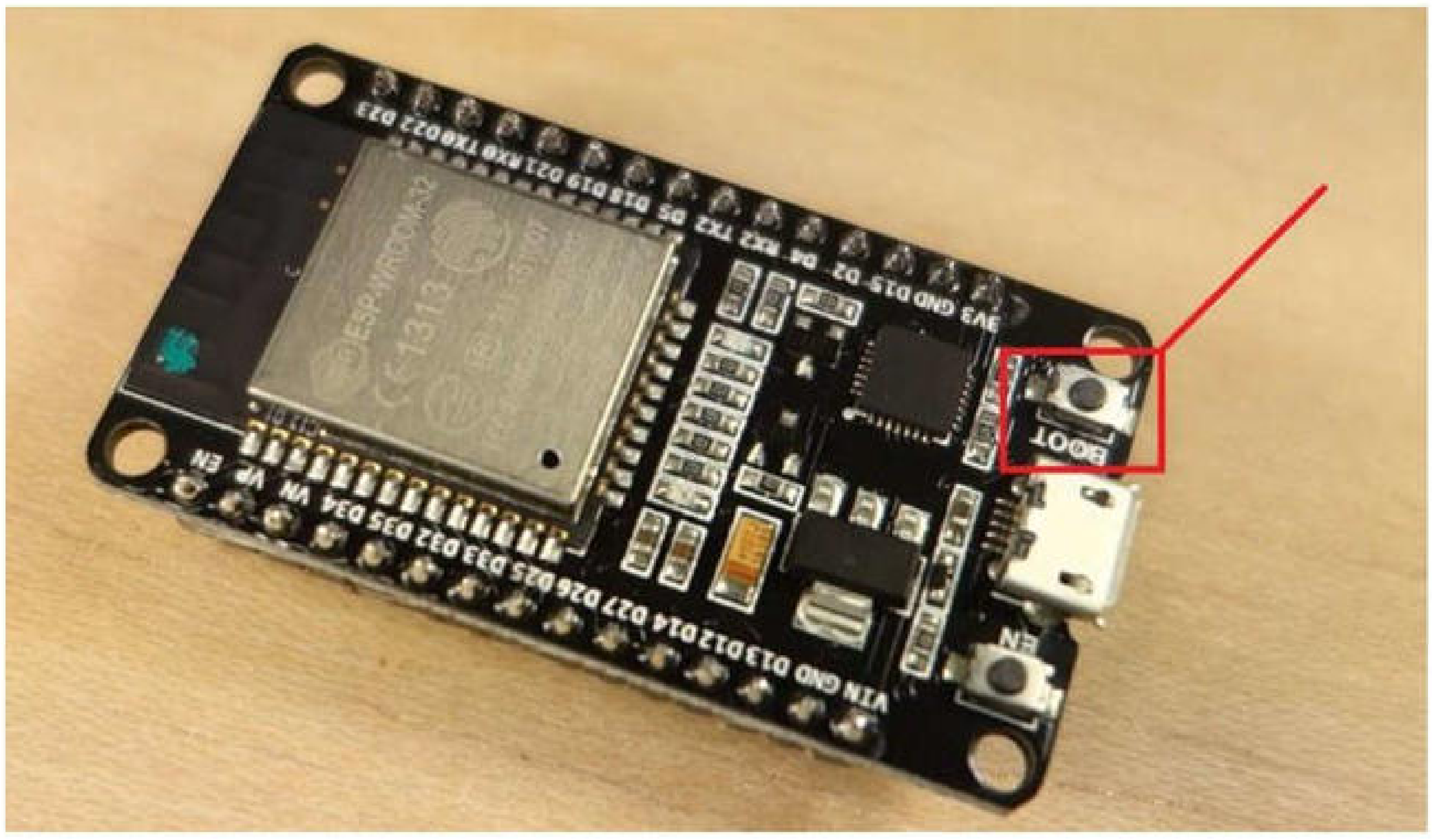
- board: **esp32**
- burn_addr: **0x1000**
- erase_flash: **yes**
- com: **COMX** (in our case it's COM5)
- Firmware: Select "**Users**" and choose the **ESP32 *.bin*** file downloaded earlier



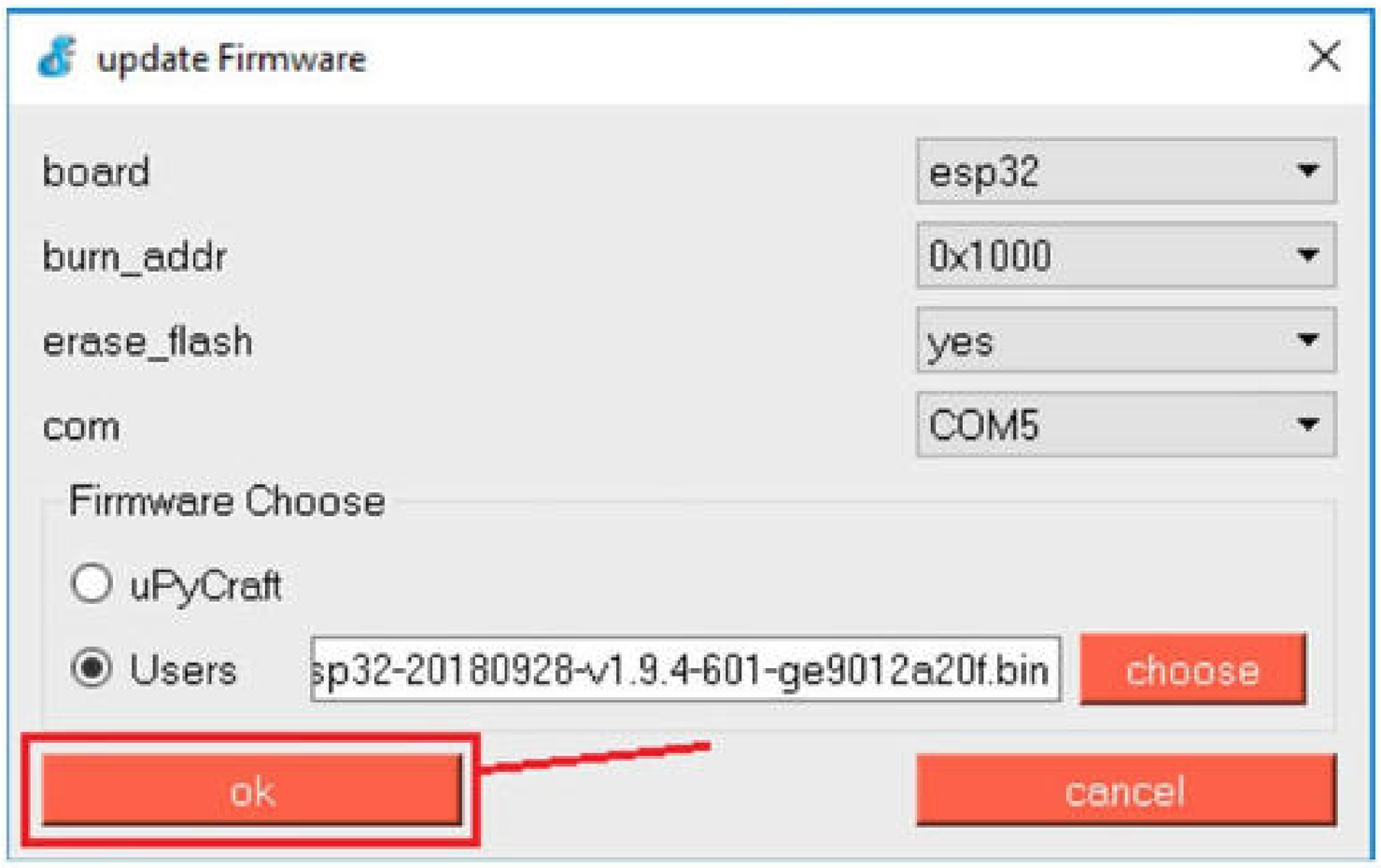
After pressing the "**Choose**" button, navigate to your *Downloads* folder and select the **ESP32 *.bin*** file you've downloaded previously:



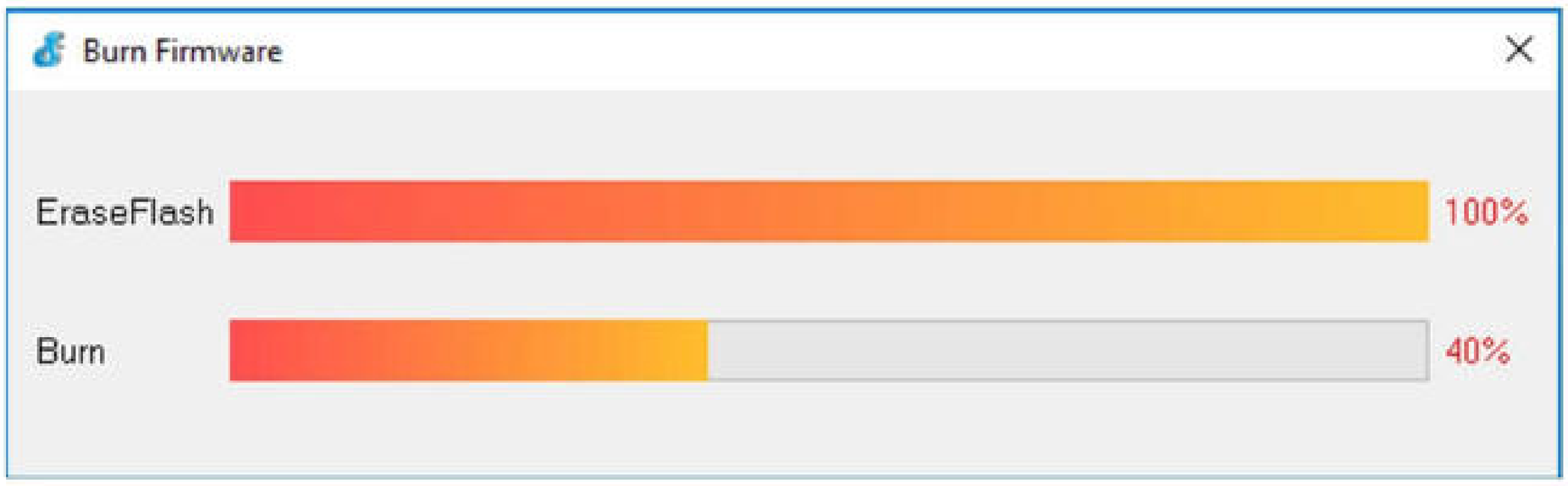
Having all the settings selected, hold-down the "**BOOT/FLASH**" button in your ESP32 board:



While holding down the "**BOOT/FLASH**", click the "**ok**" button in the burn firmware window:



When the "**EraseFlash**" process begins, you can release the "**BOOT/FLASH**" button. After a few seconds, the firmware is flashed into your ESP32 board.

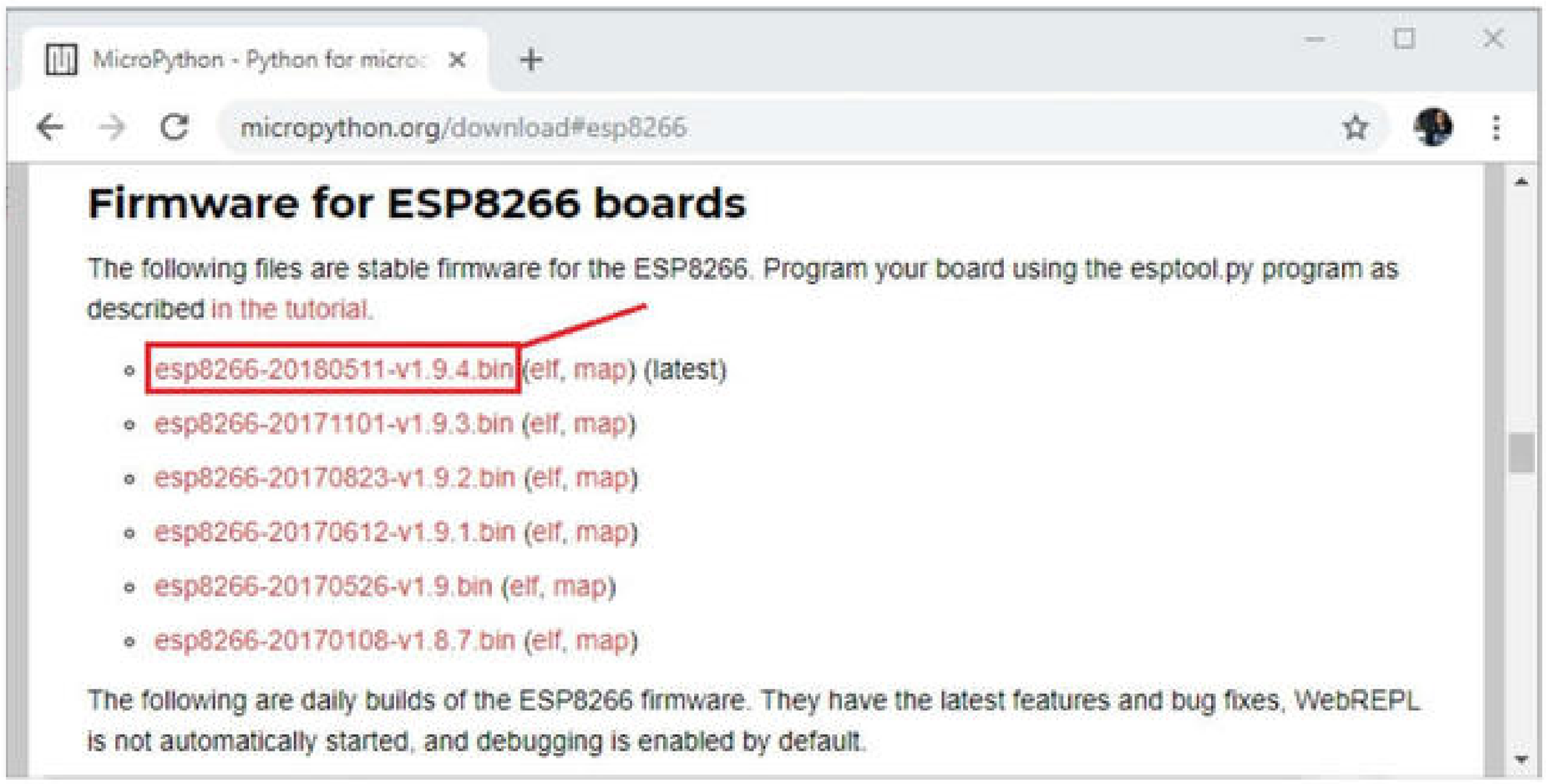


Note: if the "**EraseFlash**" bar doesn't move and you see an error message saying "**erase false**.", it means that your ESP32 wasn't in flashing mode. You need to repeat all the steps described earlier and hold the "**BOOT/FLASH**" button again to ensure that your ESP32 goes into flashing mode.

[Part 2 – ESP8266] Downloading and Flashing the MicroPython Firmware on ESP8266

To download the latest version of MicroPython firmware for the ESP8266, go to the MicroPython Downloads page and scroll all the way down to the ESP8266 section.

You should see a similar web page (see figure below) with the latest link to download the ESP8266 *.bin file* – for example: ***esp8266-20180511-v1.9.4.bin***.



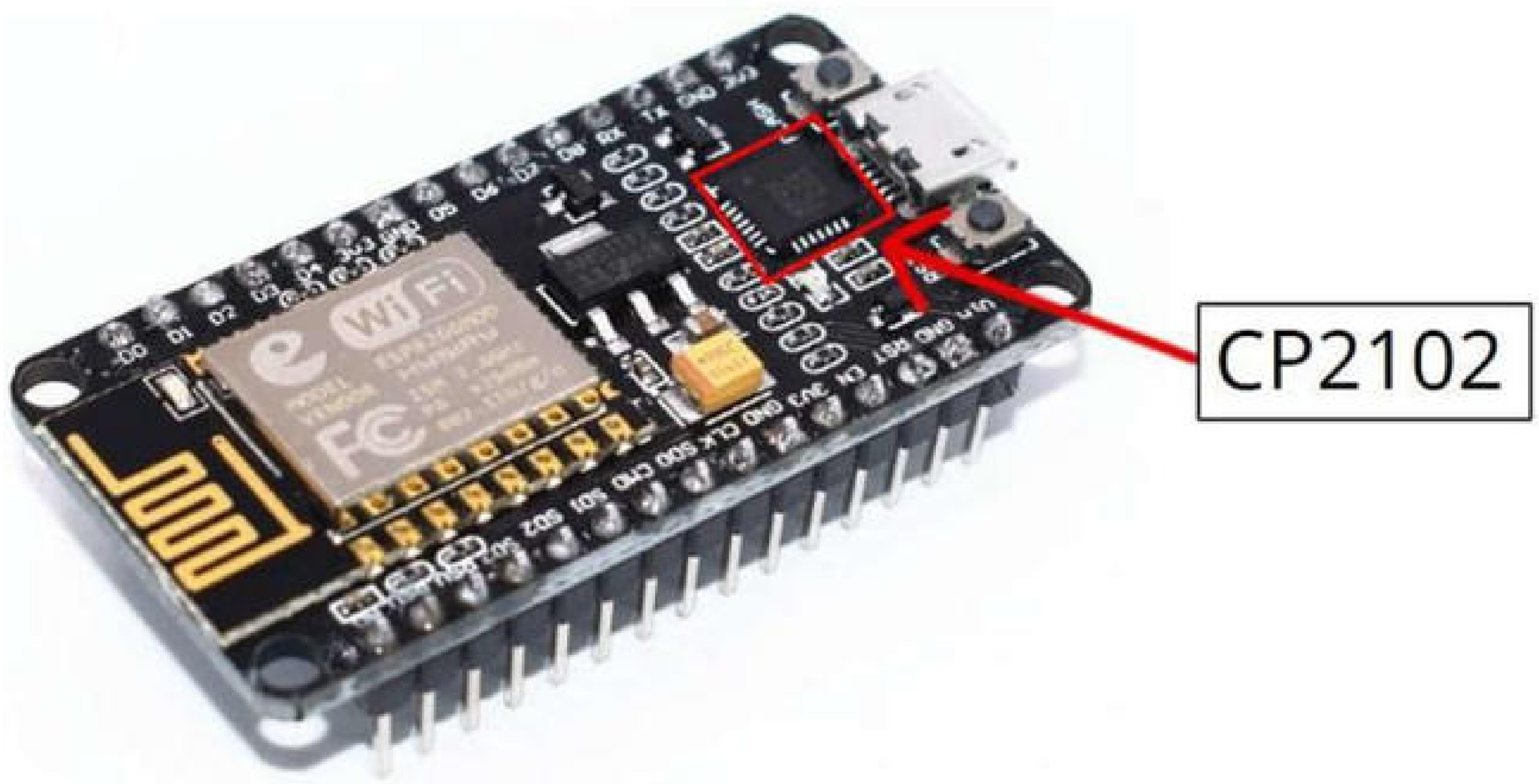
Selecting Serial Port

Connect your ESP8266 to your computer. Go to **Tools** ▸ **Serial** and select your ESP8266 COM port (in our case it's COM5).

IMPORTANT: if you plug your ESP32 board to your computer, but you can't find the ESP32 Port available in your uPyCraft IDE, it might be one of these two problems: **1.** USB drivers missing or **2.** USB cable without data wires.

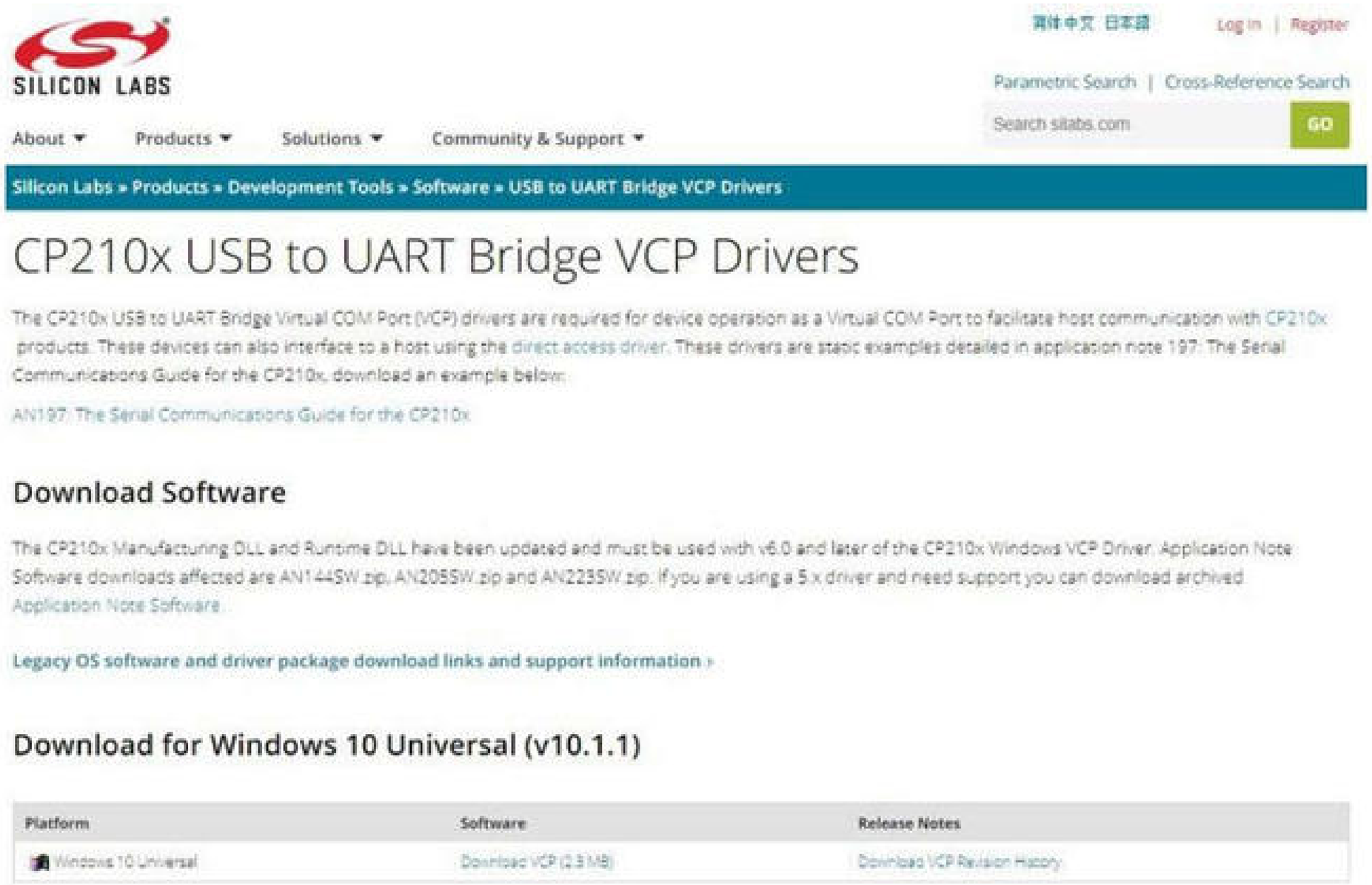
1. If you don't see your ESP's COM port available, this often means you don't have the USB drivers installed. Take a closer look at the chip next to the voltage regulator on board and check its name.

The ESP8266 ESP-12E NodeMCU board uses the CP2102 chip.



Go to Google and search for your specific chip to find the drivers and install them in your operating system.

You can download the CP2102 drivers on the Silicon Labs website.



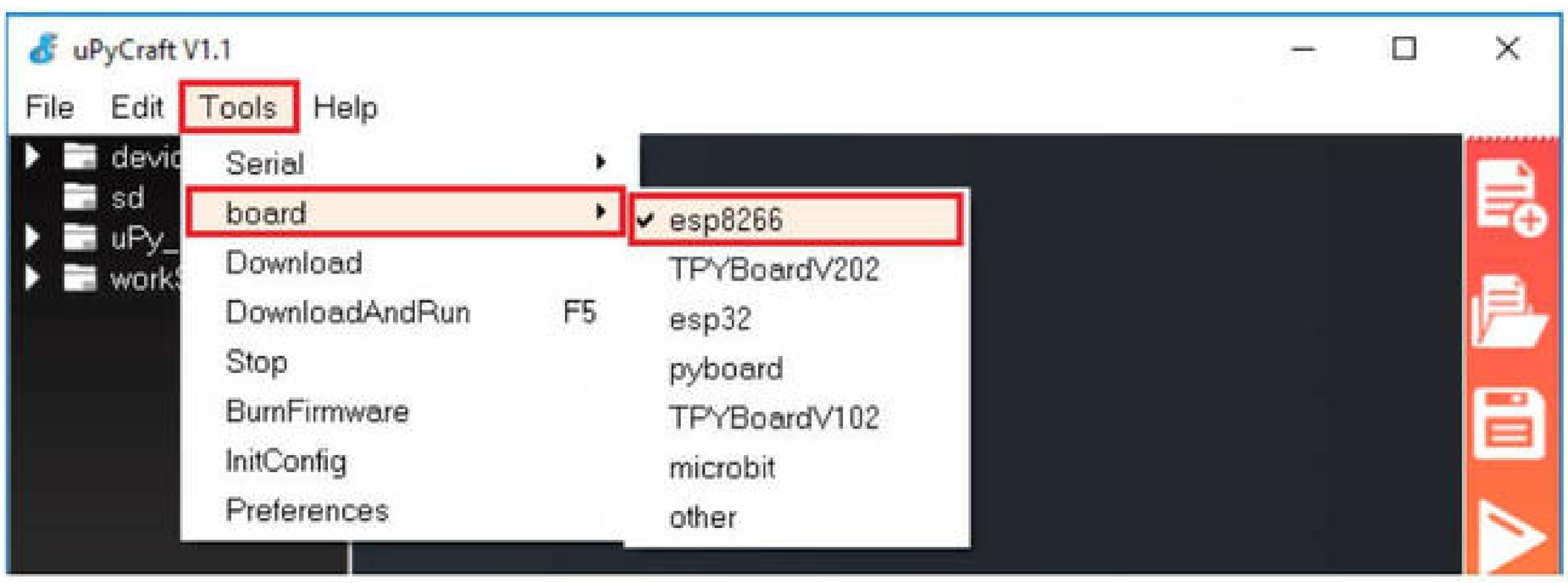
After they are installed, restart the uPyCraft IDE and you should see the COM port in the **Tools** menu.

2. If you have the drivers installed, but you can't see your device, double-check that you're using a USB cable with data wires.

USB cables from power banks often don't have data wires (they are charge only). So, your computer will never establish a serial communication with your ESP8266. Using a proper USB cable should solve your problem.

Selecting the Board

Go to **Tools** ▸ **Board**. For this tutorial, we assume that you're using the ESP8266, so make sure you select the "**esp8266**" option:

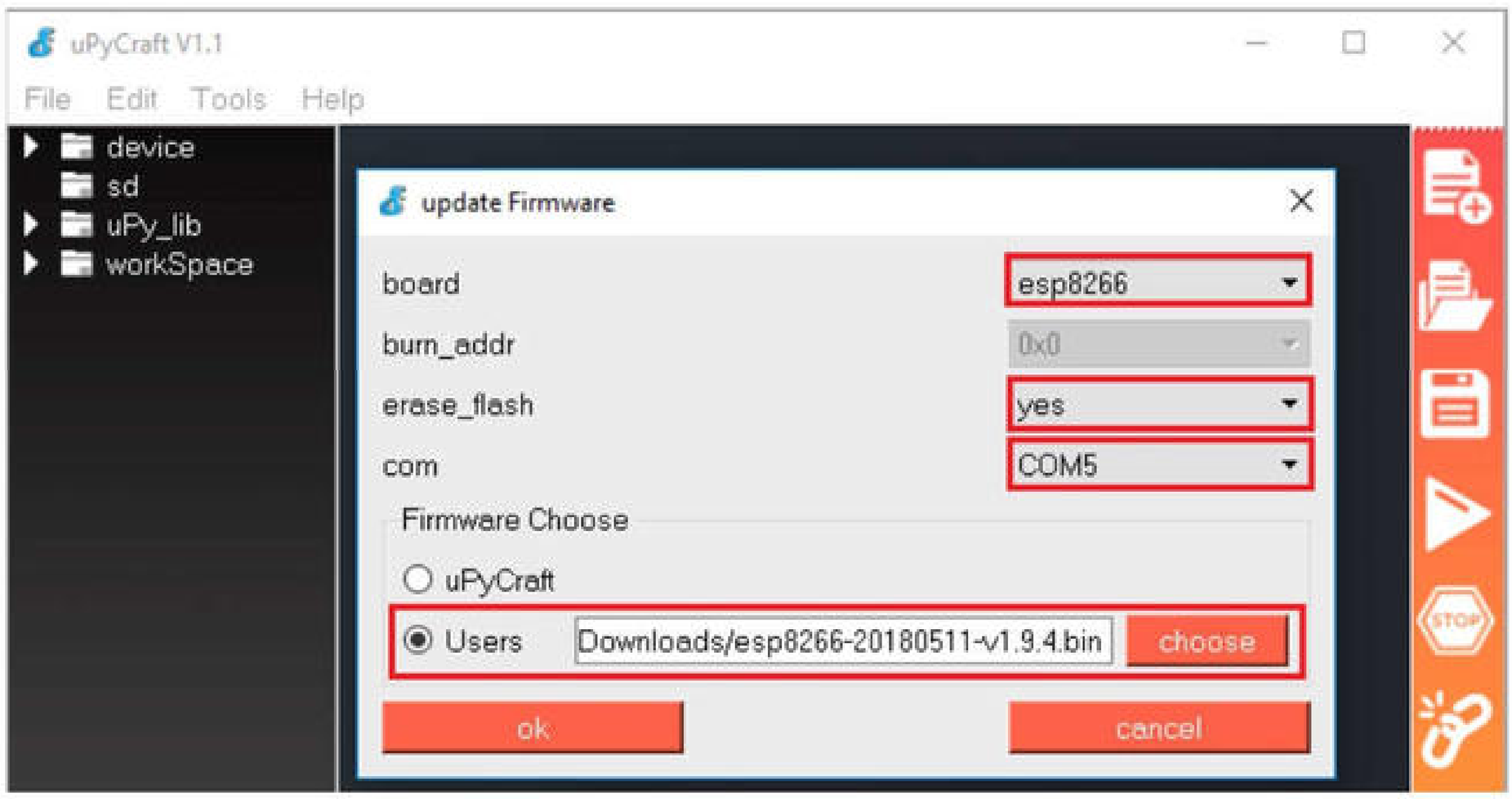


Flashing/Uploading MicroPython Firmware

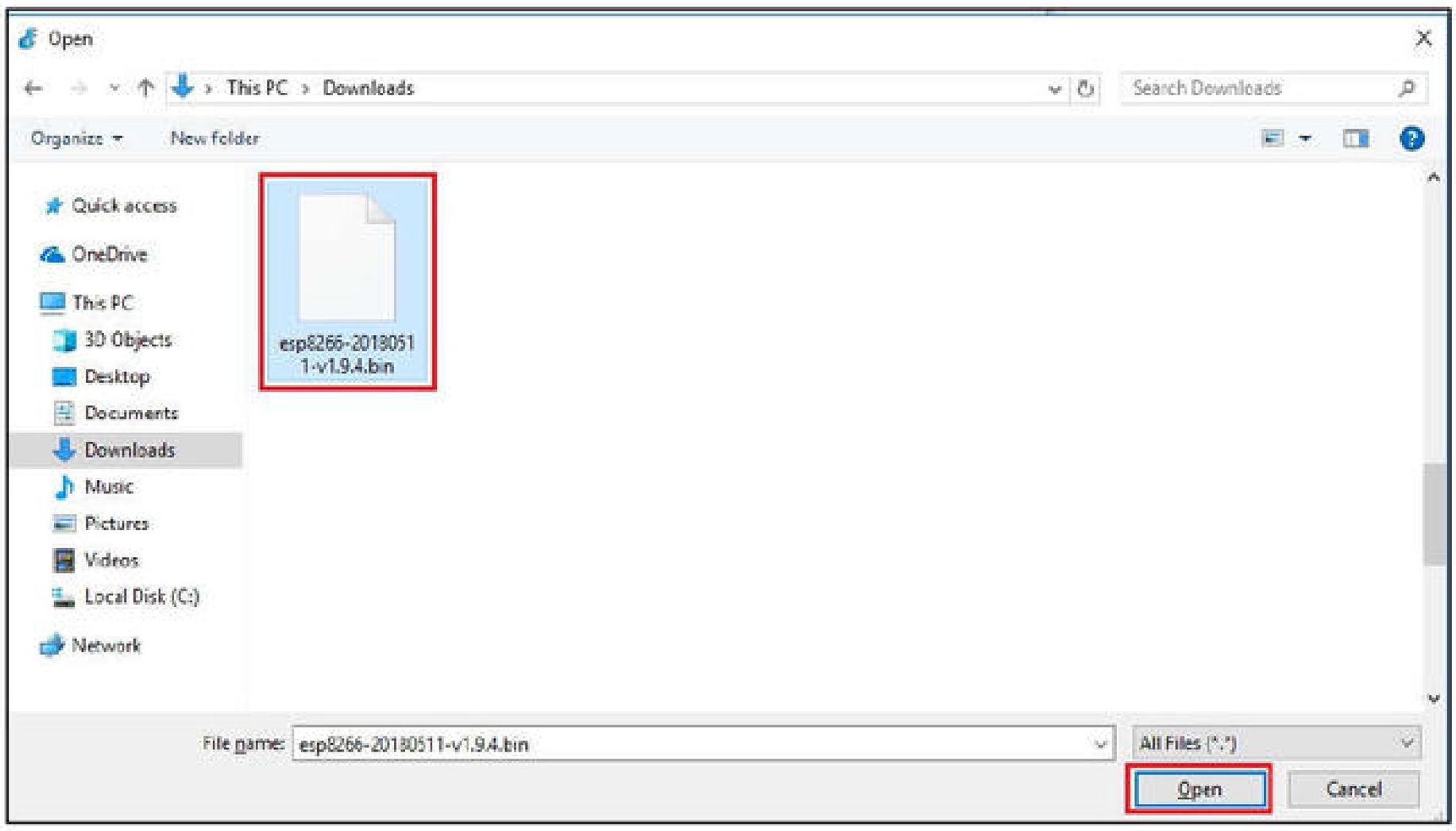
Finally, go to **Tools** ▸ **BurnFirmware** menu to flash your ESP32 with MicroPython.

Select all these options to flash the ESP8266 board:

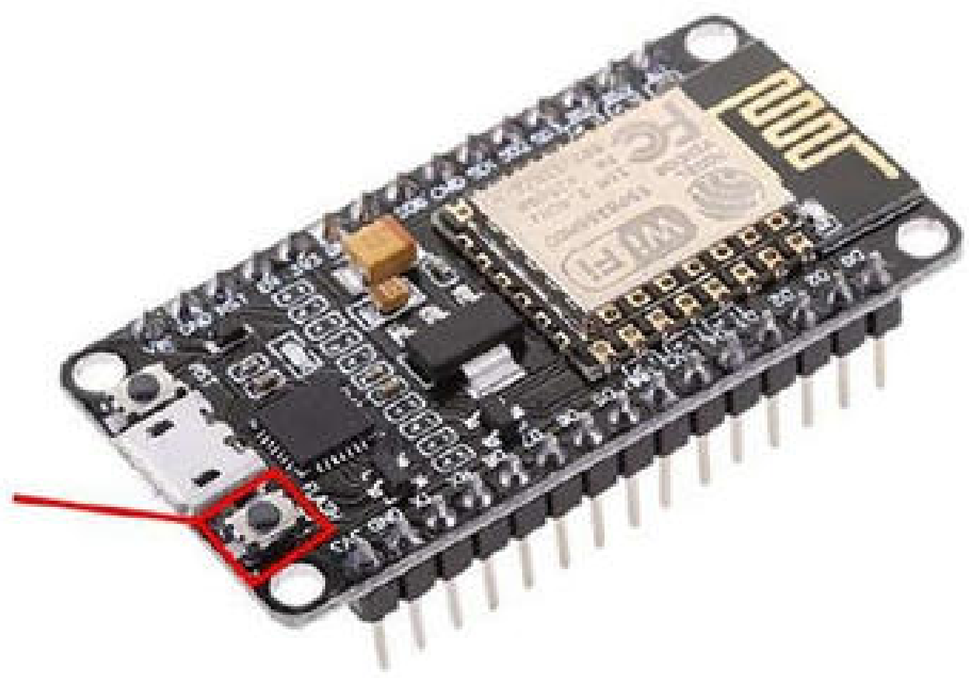
- board: **esp8266**
- burn_addr: **0x0**
- erase_flash: **yes**
- com: **COMX** (in our case it's COM5)
- Firmware: Select "**Users**" and choose the ESP8266 ***.bin*** file downloaded earlier



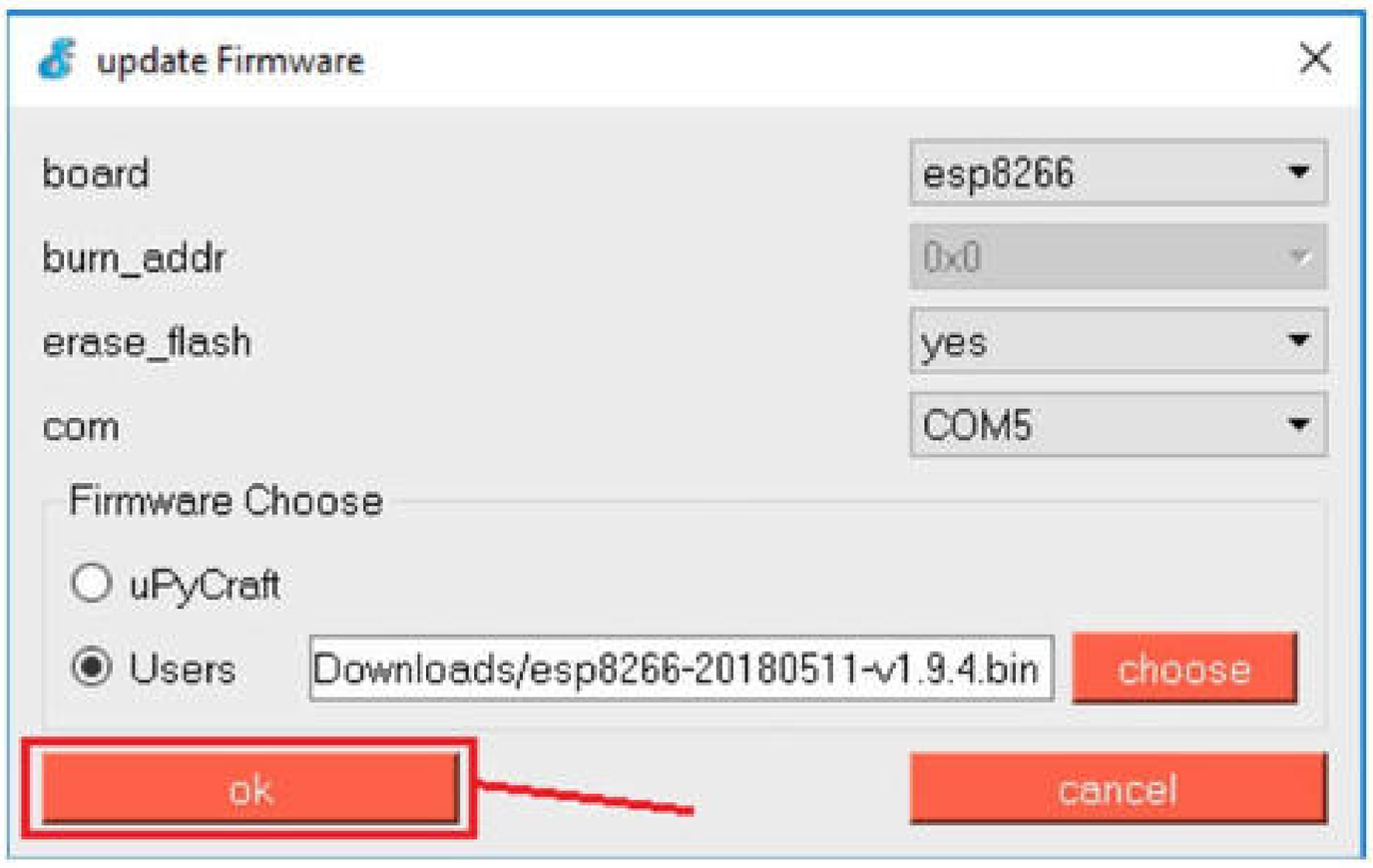
After pressing the "**Choose**" button, navigate to your Downloads folder and select the ESP8266 ***.bin*** file you've downloaded earlier:



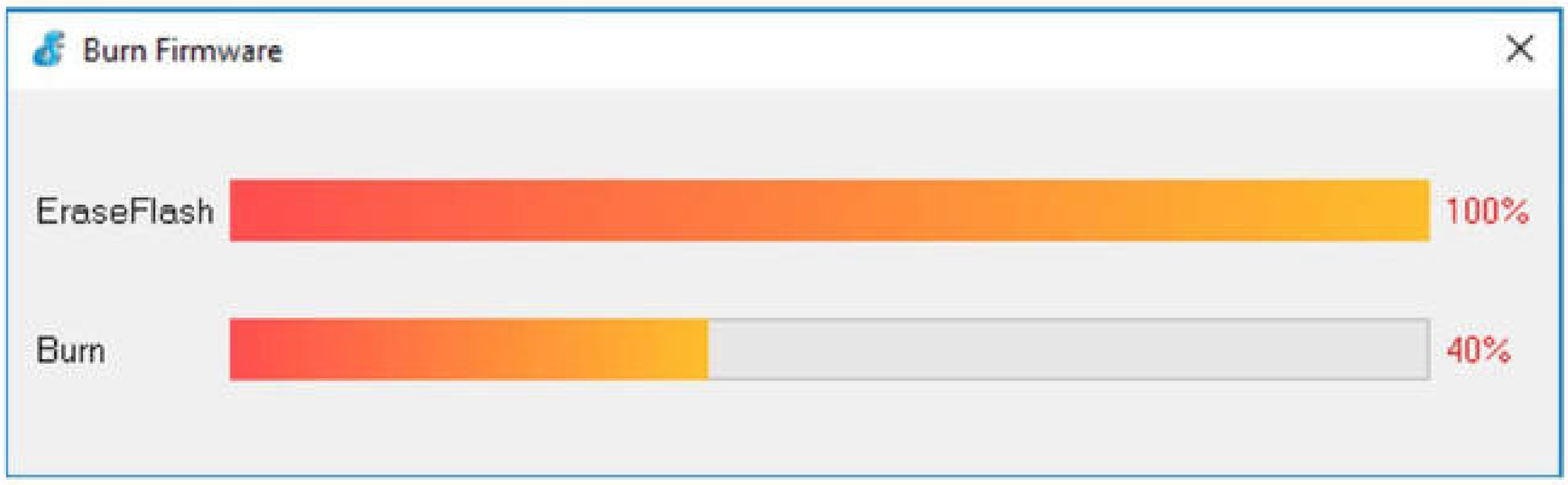
Having all the settings selected, hold-down the "**BOOT/FLASH**" button in your ESP8266 board:



While holding down the "**BOOT/FLASH**", click the "**ok**" button in the burn firmware window:

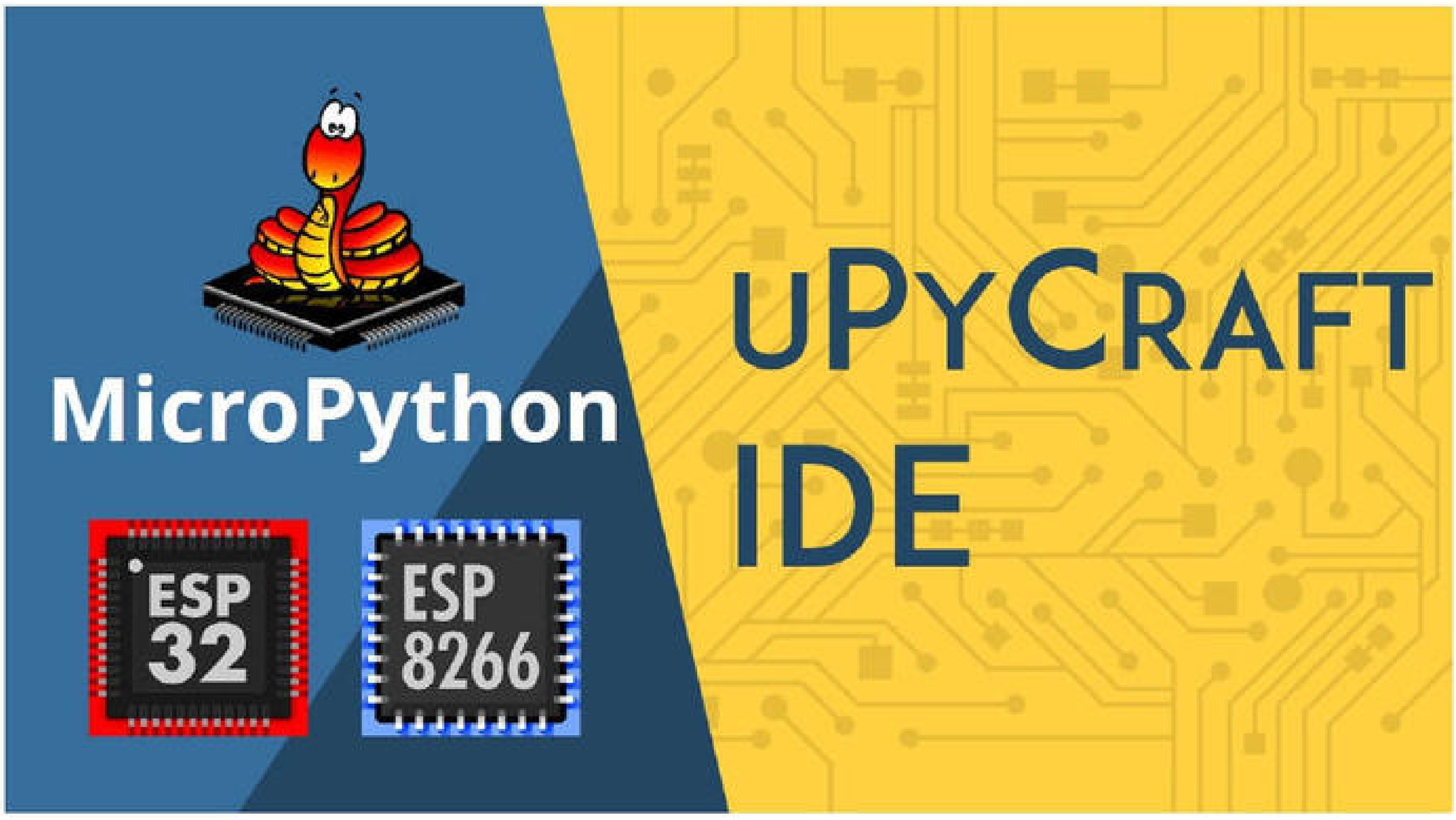


When the "**EraseFlash**" process begins, you can release the "**BOOT/FLASH**" button. After a few seconds, the firmware is flashed into your ESP8266 board.



Note: if the "**EraseFlash**" bar doesn't move and you see an error message saying "**erase false**.", it means that your ESP8266 wasn't in flashing mode. You need to repeat all the steps described earlier and hold the "**BOOT/FLASH**" button again to ensure that your ESP8266 goes into flashing mode.

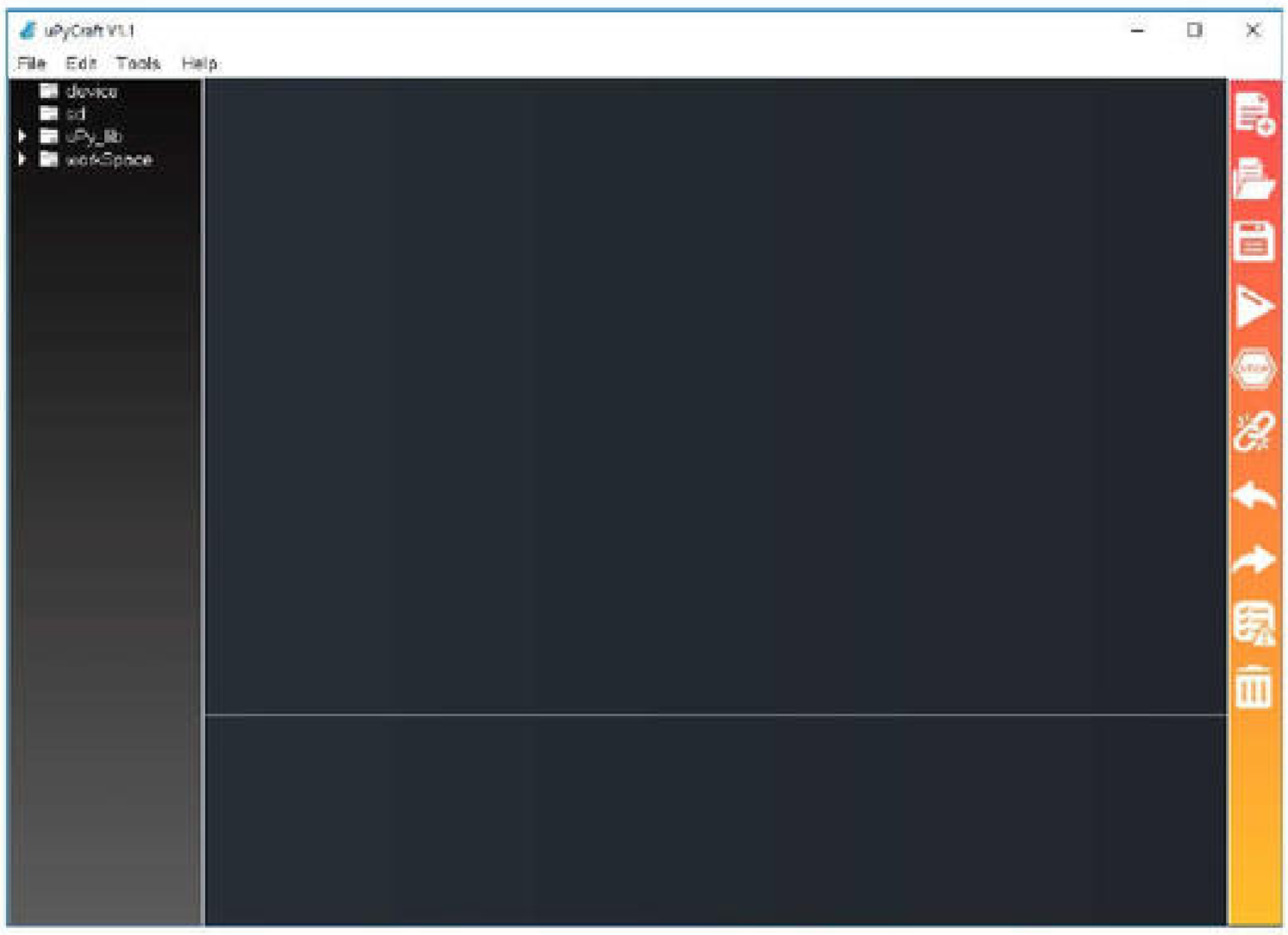
Getting Started with uPyCraft IDE



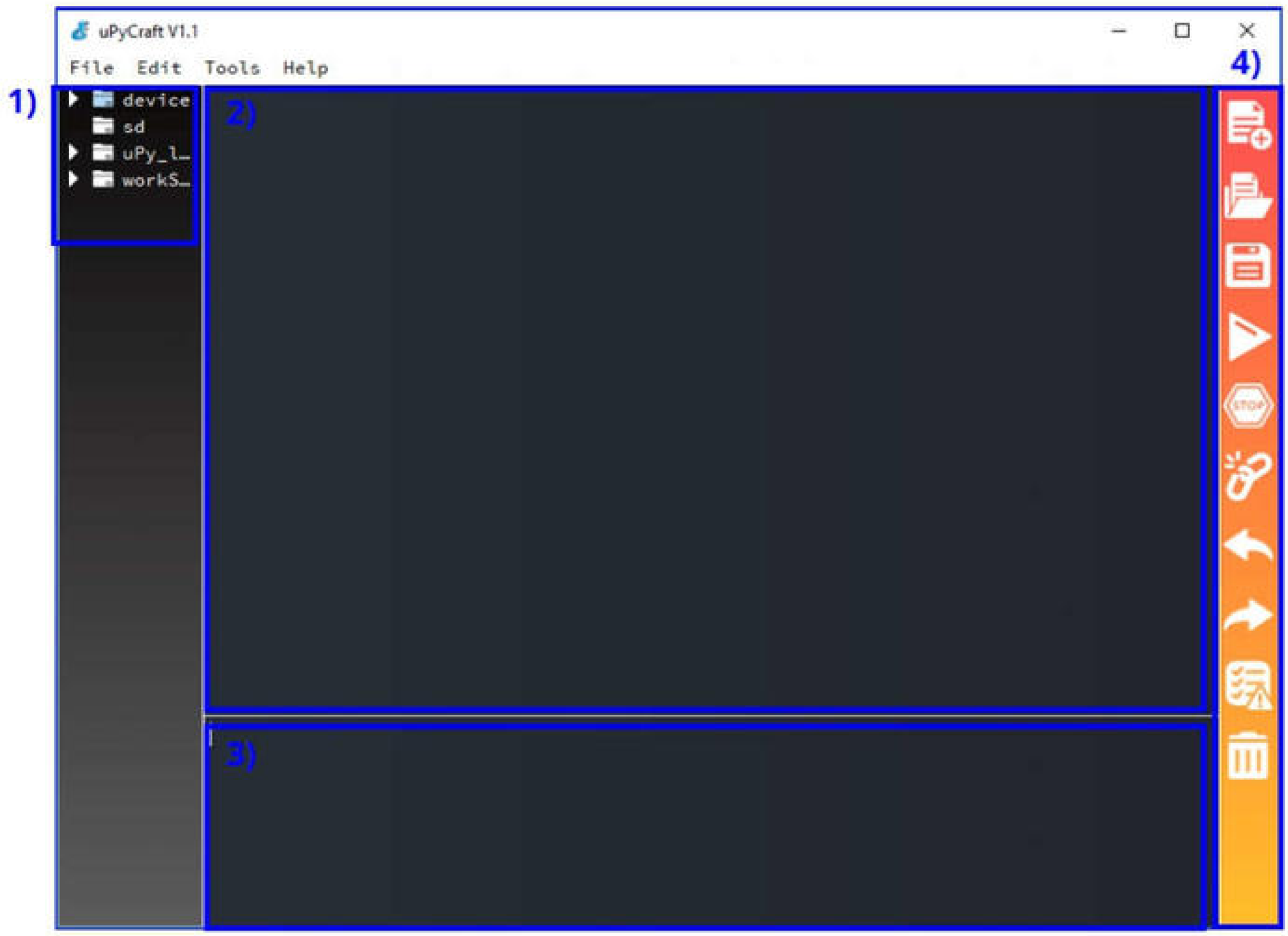
In this section, we'll give you an overview of the uPyCraft IDE software, so that you can start programming the ESP32/ESP8266 with MicroPython.

uPyCraft IDE Overview

At this point, we assumed that you have uPyCraft IDE installed on your computer and the ESP32/ESP8266 flashed with MicroPython firmware. Open uPyCraft IDE, a new window opens as follows:



Let's take a closer look at each section of uPyCraft IDE:



1. Folder and files
2. Editor
3. MicroPython Shell/Terminal
4. Tools

1. Folder and files

This section shows several folders and files. The *device* folder shows the files that are currently stored on your ESP board. If you have your ESP32 or ESP8266 connected via serial to uPyCraft IDE, when you expand the *device* folder, all files stored in the ESP32 or ESP8266 should load. By default, you should only have a *boot.py* file. To run your main code, we recommend creating a *main.py* file.

- ***boot.py***: runs when the device starts and sets up several configuration options;
- ***main.py***: this is the main script that contains your code. It executes immediately after the *boot.py*.

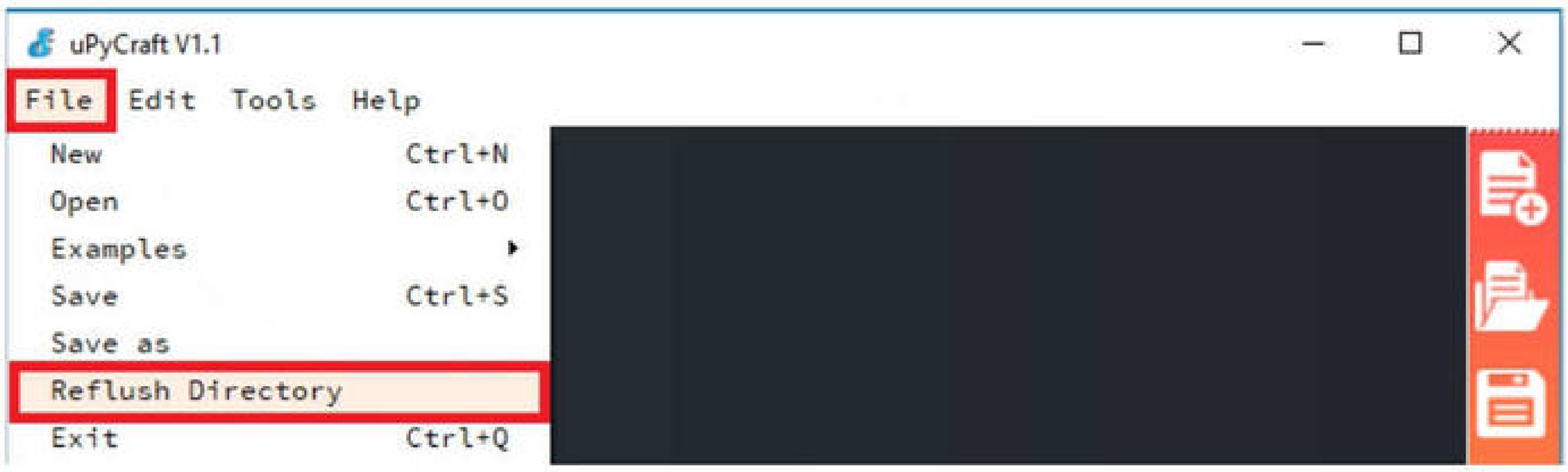
The *sd* folder is meant to access files stored on SD cards – this only works with boards like the PyBoard that come with an SD card slot.

The *uPy_lib* folder shows the built-in IDE library files.

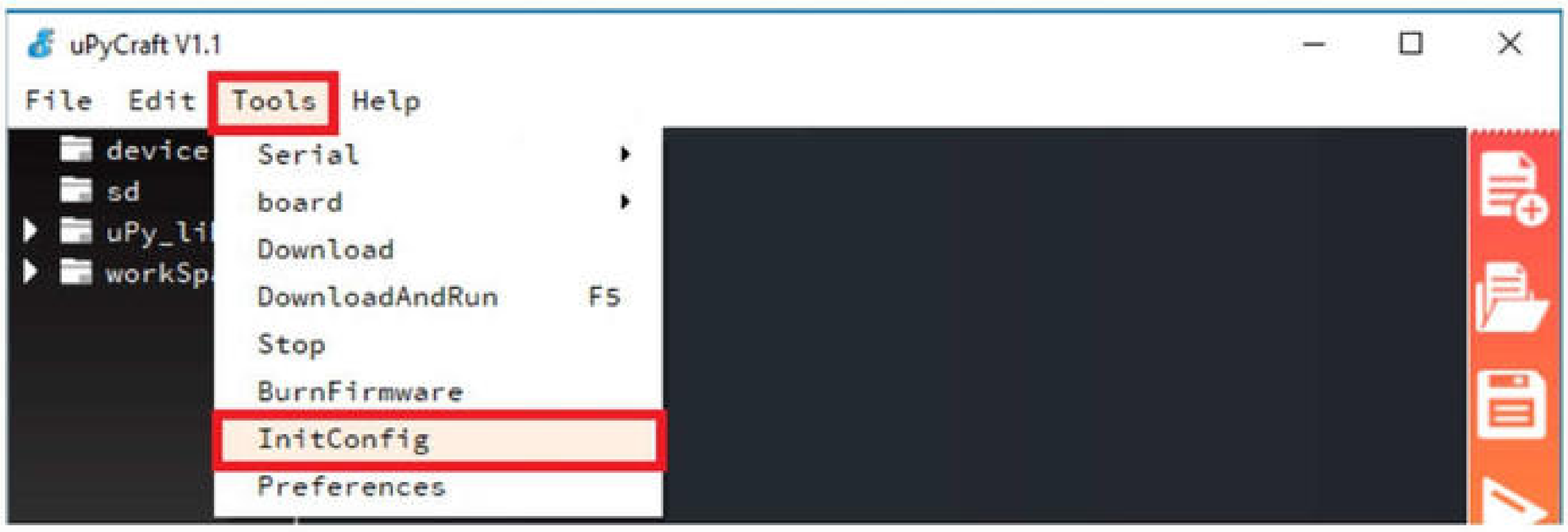
Finally, the *workspace* is a directory to save your files. These files are saved in your computer in a directory that you define. This is especially useful to keep all your files organized at hand.

When using uPyCraft IDE for the first time, to select your working directory, click the *workspace* folder. A new window pops up for you to choose your workspace path. Create a new folder or select an existing folder to be your working directory.

Then, go to **File ▸ Reflush Directory** to update the directory.



Note: to change your user directory, simply go to **Tools ▸ InitConfig** and click the **workspace** directory folder to choose a different path.



2. Editor

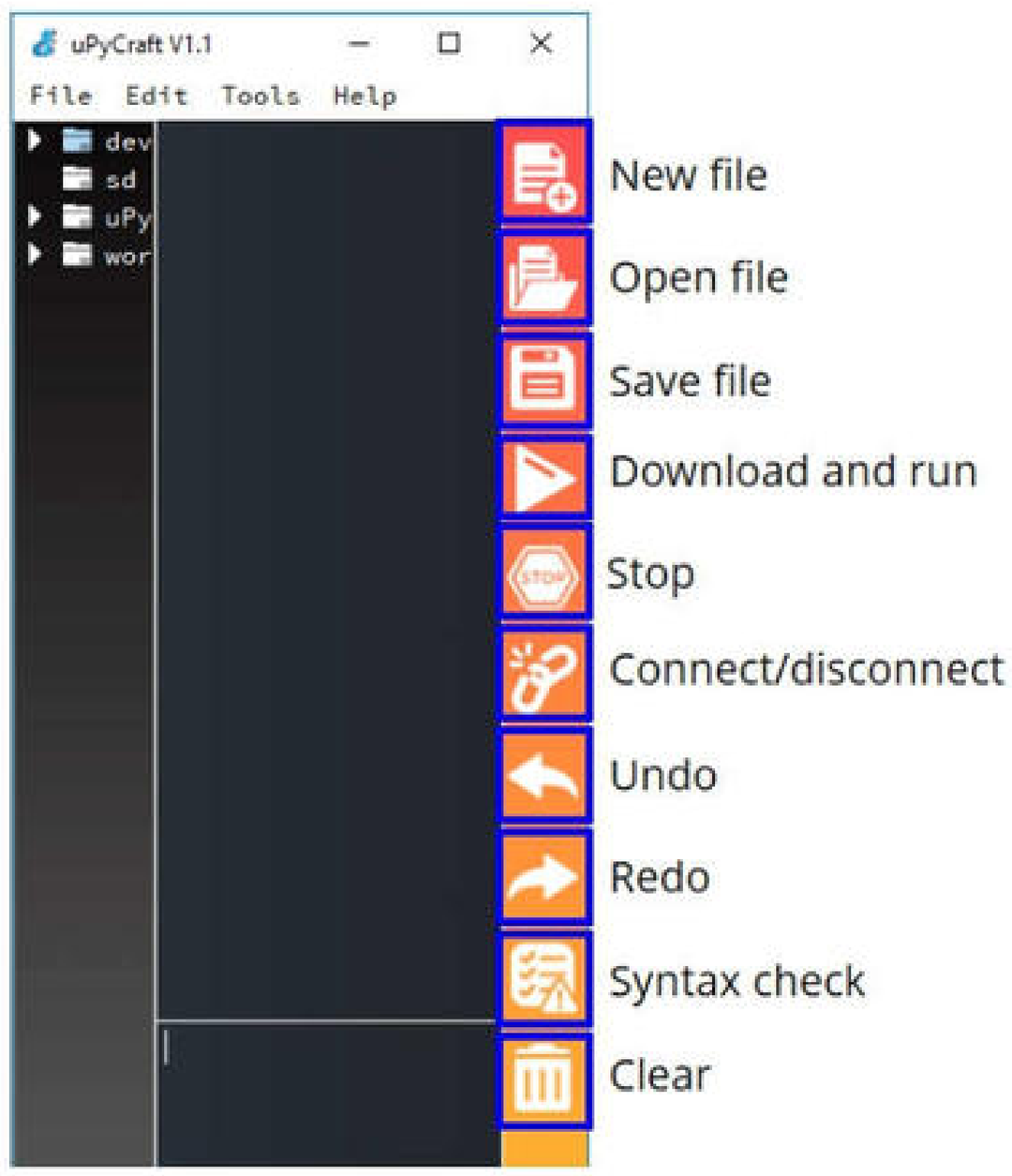
The **Editor** section is where you write your code and edit your *.py* files. You can open more than one file, and the Editor will open a new tab for each file.

3. MicroPython Shell/terminal

On the MicroPython Shell you can type commands to be executed immediately by your ESP board without the need to upload new files. The terminal also provides information about the state of an executing program, shows errors related with upload, syntax errors, prints messages, etc...

4. Tools

The icons placed at the rightmost side allow you to quickly perform tasks. Each button is labeled in the figure below:



- **New file**: creates a new file on the Editor;
- **Open file**: opens a file from your computer;
- **Save file**: saves a file;
- **Download and run**: upload the code to your board and execute the code;
- **Stop**: stop the execution of the code – it's the same as entering CRTL+C on the Shell to stop all scripts from running;
- **Connect/Disconnect**: connect or disconnect to your board via Serial. You must select the serial port first in **Tools ▸ Serial**;
- **Undo**: undo last change in the code Editor;

- **Redo**: redo last change in the code Editor;
- **Syntax check**: checks the syntax of your code;
- **Clear**: clear the Shell/terminal window messages.

Running Your First Script

To get you familiar with the process of writing a file and executing code on your ESP32/ESP8266 boards, we'll upload a new script that simply blinks the on-board LED of your ESP32 or ESP8266.

Establishing a communication with the board

After having the MicroPython firmware installed on your board and having the board connected to your computer through a USB cable, follow the next steps:

1. Go to **Tools** ▸ **Board** and select the board you're using.

2. Go to **Tools** ▸ **Port** and select the COM port your ESP is connected to.

3. Press the **Connect** button to establish a serial communication with your board.

Connect/disconnect

4. The **>>>** should appear in the Shell window after a successful connection with your board. You can type the `print` command to test if it's working:

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

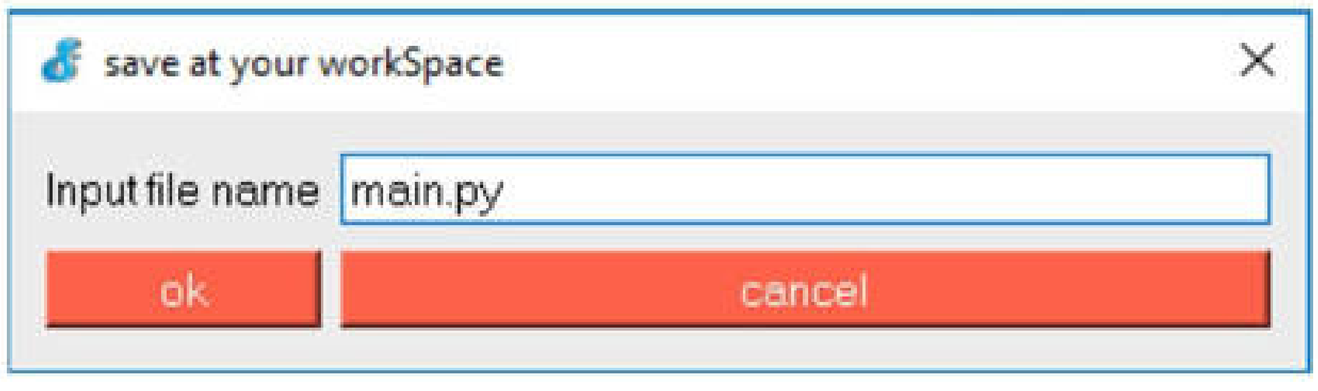
It should print the "**Hello**" message. Only if you see that message, you can continue with this tutorial. Otherwise, make sure you have established a serial communication with your board or that you've successfully flashed the MicroPython firmware on your board.

Creating the *main.py* file on your board

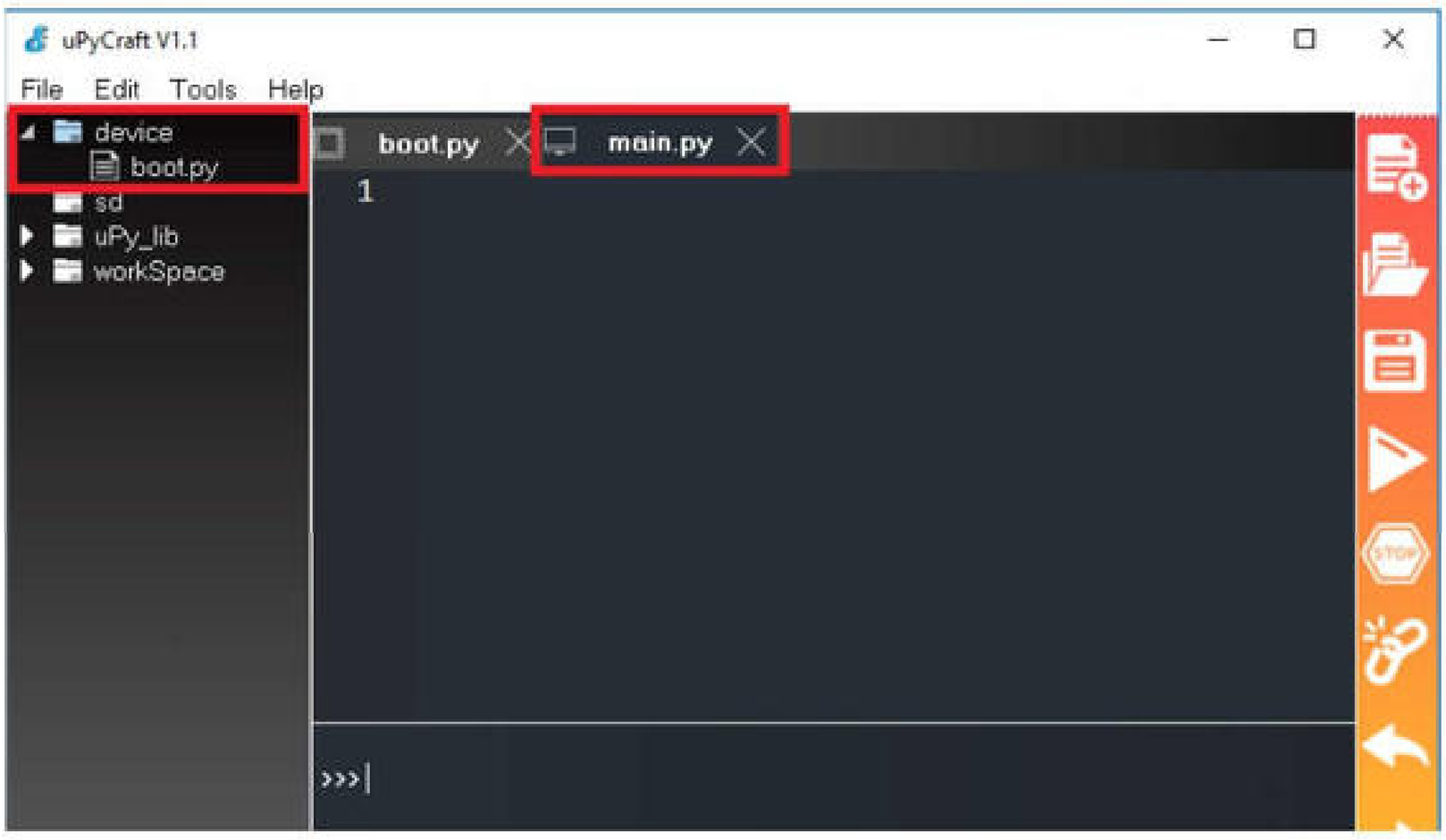
1. Press the "**New file**" button to create a new file.

2. Press the "**Save file**" button to save the file in your computer.

3. A new window opens, name your file ***main.py*** and save it in your computer:

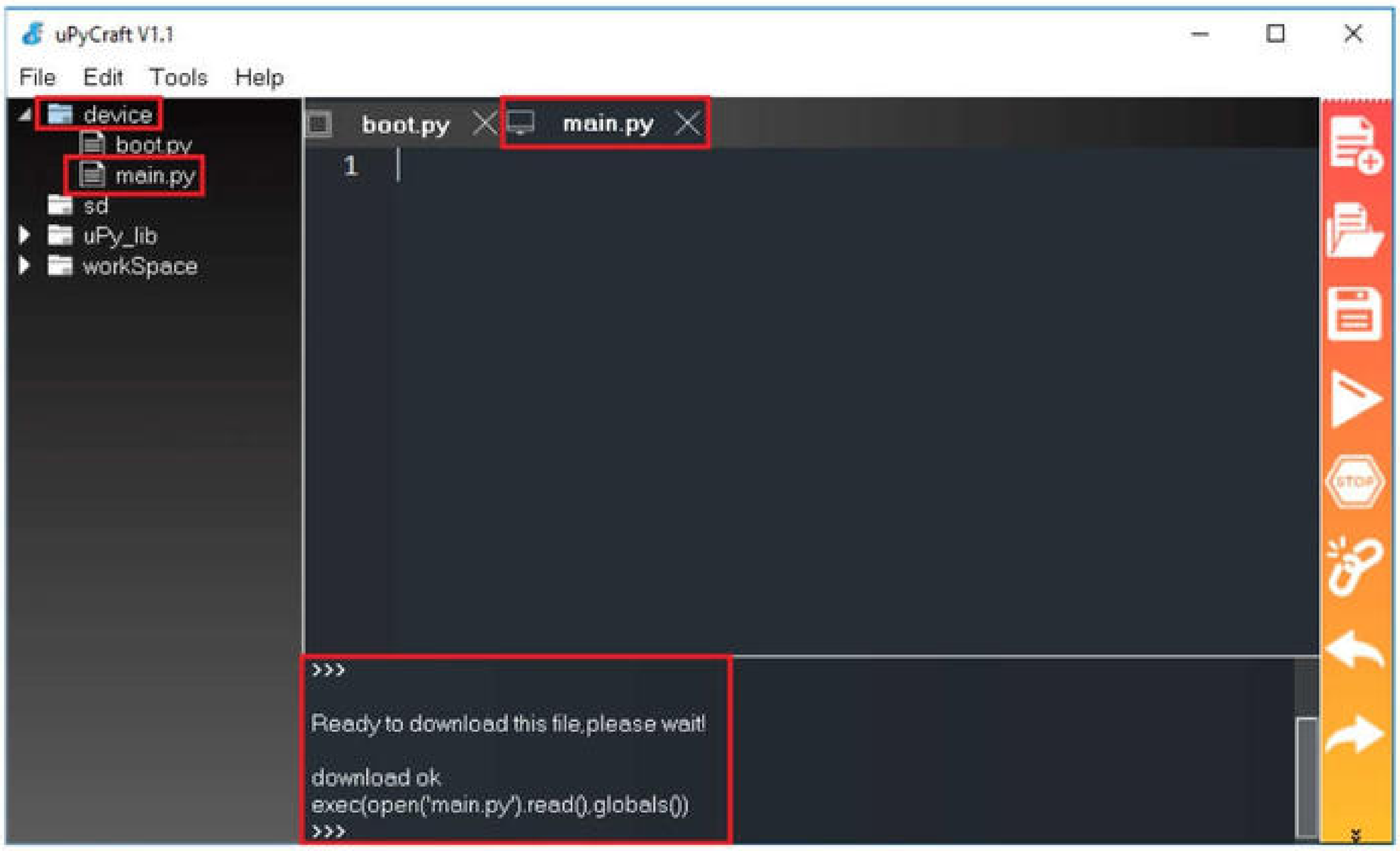


4. After that, you should see the following in your uPyCraft IDE (the *boot.py* file in your device and a new tab with the *main.py* file):



5. Click the "**Download and run**" button to upload the file to your ESP board:

6. The **device** directory should load the ***main.py*** file.



Uploading the blink LED script

1. Copy the following code to the Editor on the ***main.py*** file:

```
from machine import Pin
from time import sleep

led = Pin(2, Pin.OUT)

while True:
  led.value(not led.value())
  sleep(0.5)
```

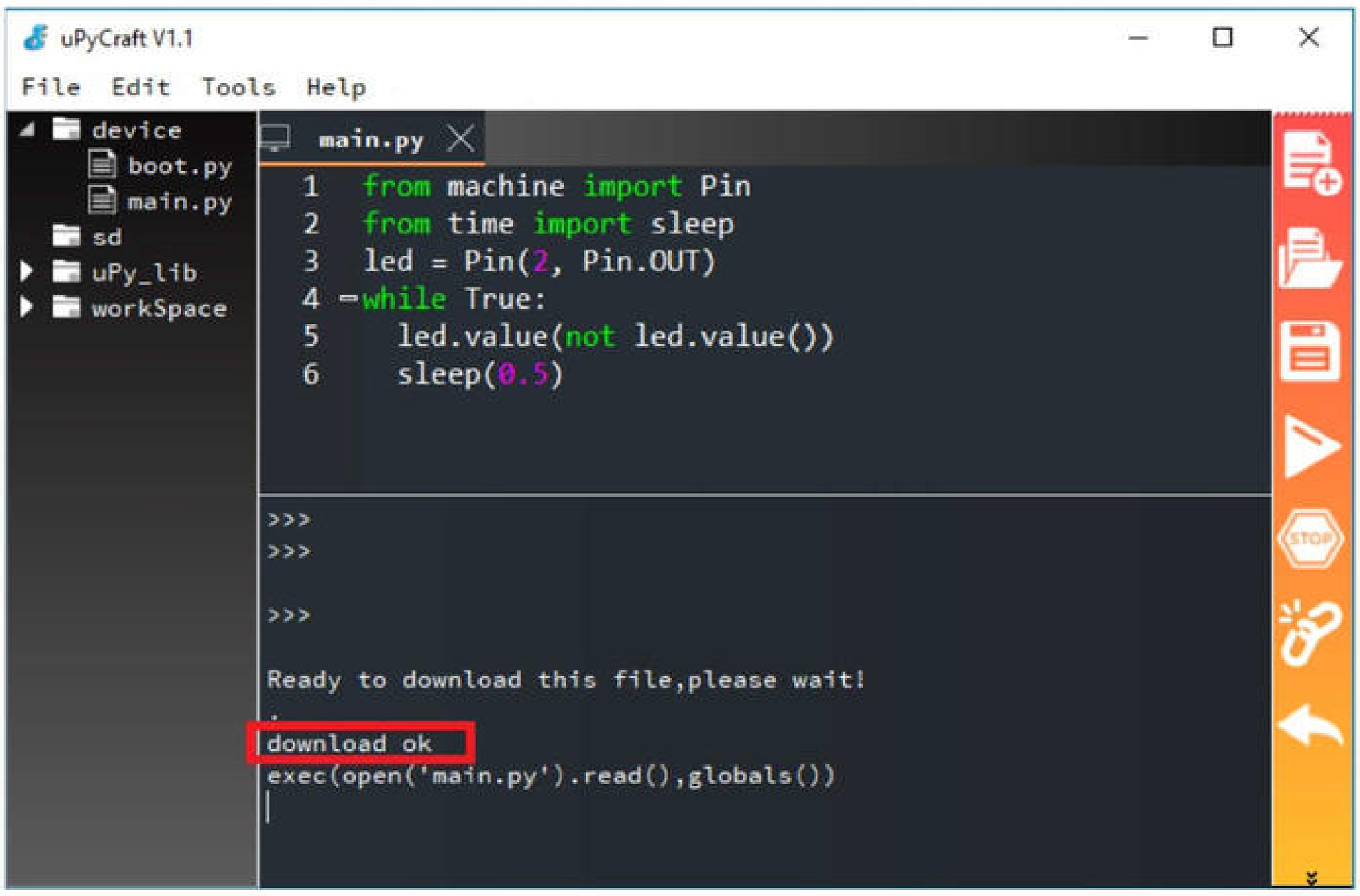
SOURCE CODE
https://github.com/RuiSantosdotme/ESP-MicroPython/blob/master/code/Blink_LED/Blink_LED.py

Note: you can click the link in the SOURCE CODE box to see the code on GitHub and copy it to uPyCraft IDE.

2. Press the "**Stop**" button to stop any script from running in your board:

3. Click the "**Download and Run button**" to upload the script to the board:

4. You should see a message saying "**download ok**" in the Shell window.

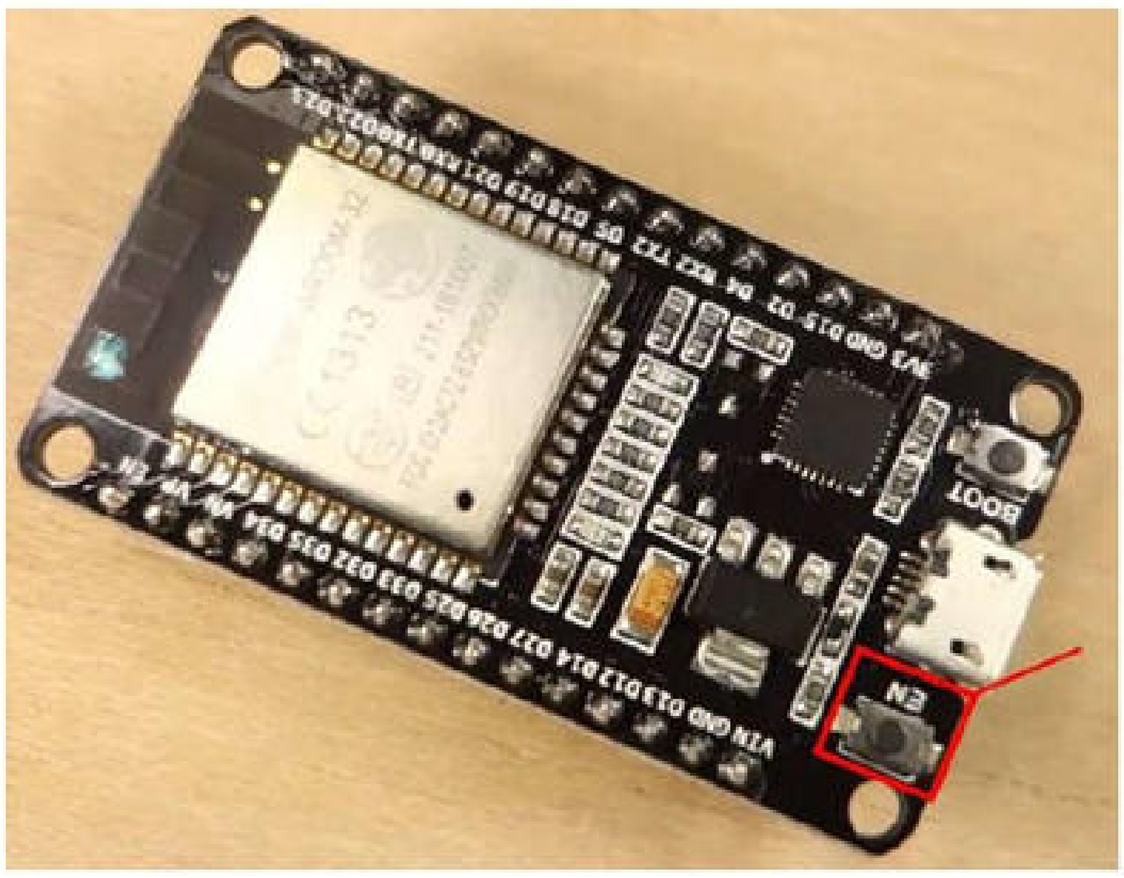


Testing the script

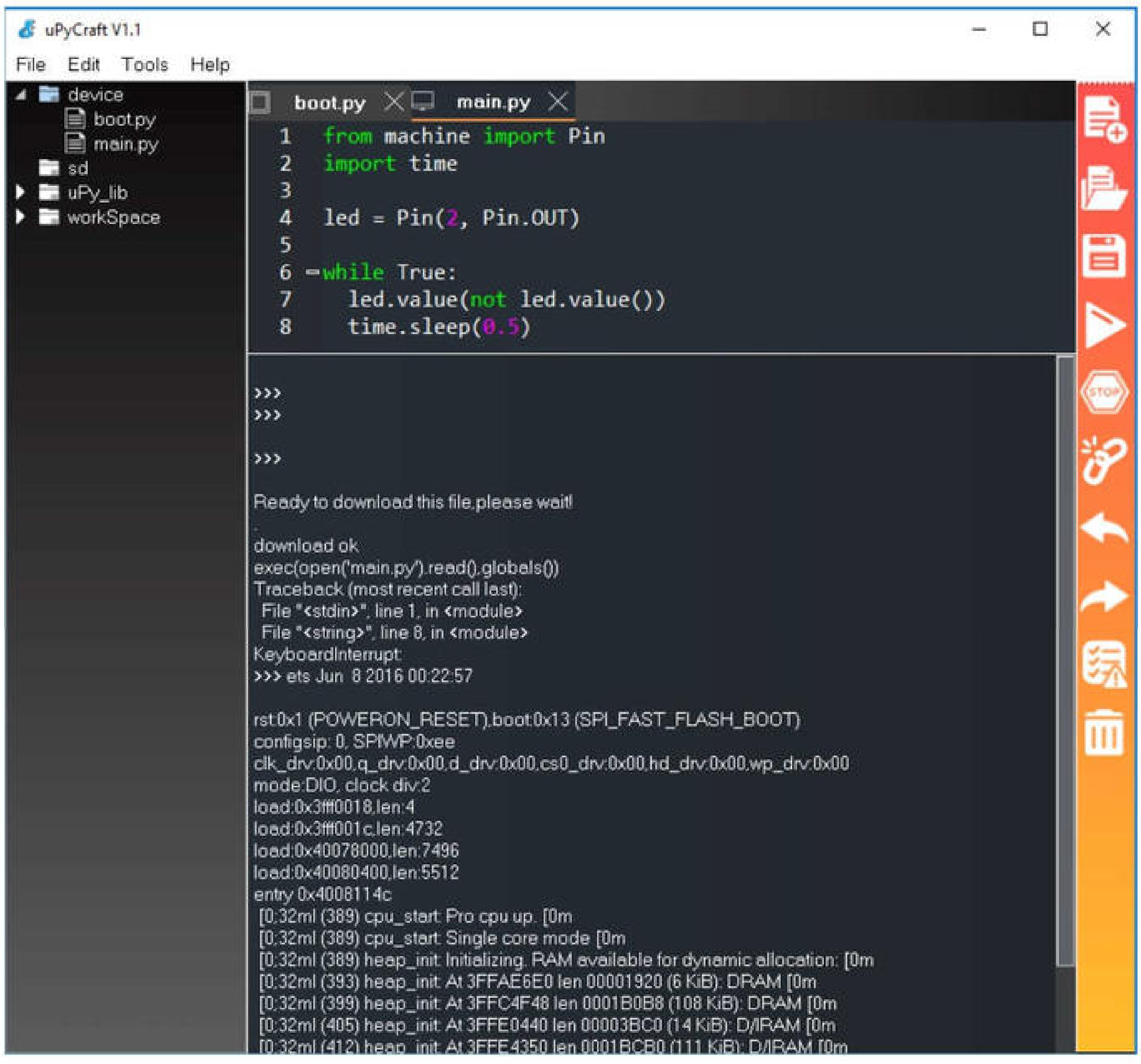
To run the script that was just uploaded to your board, you need to follow these steps:

1. Press the "**Stop**" button

2. Press the on-board ESP32/ESP8266 **EN** (**ENABLE**) or **RST** (**RESET**) button to restart your board and run the script from the start:

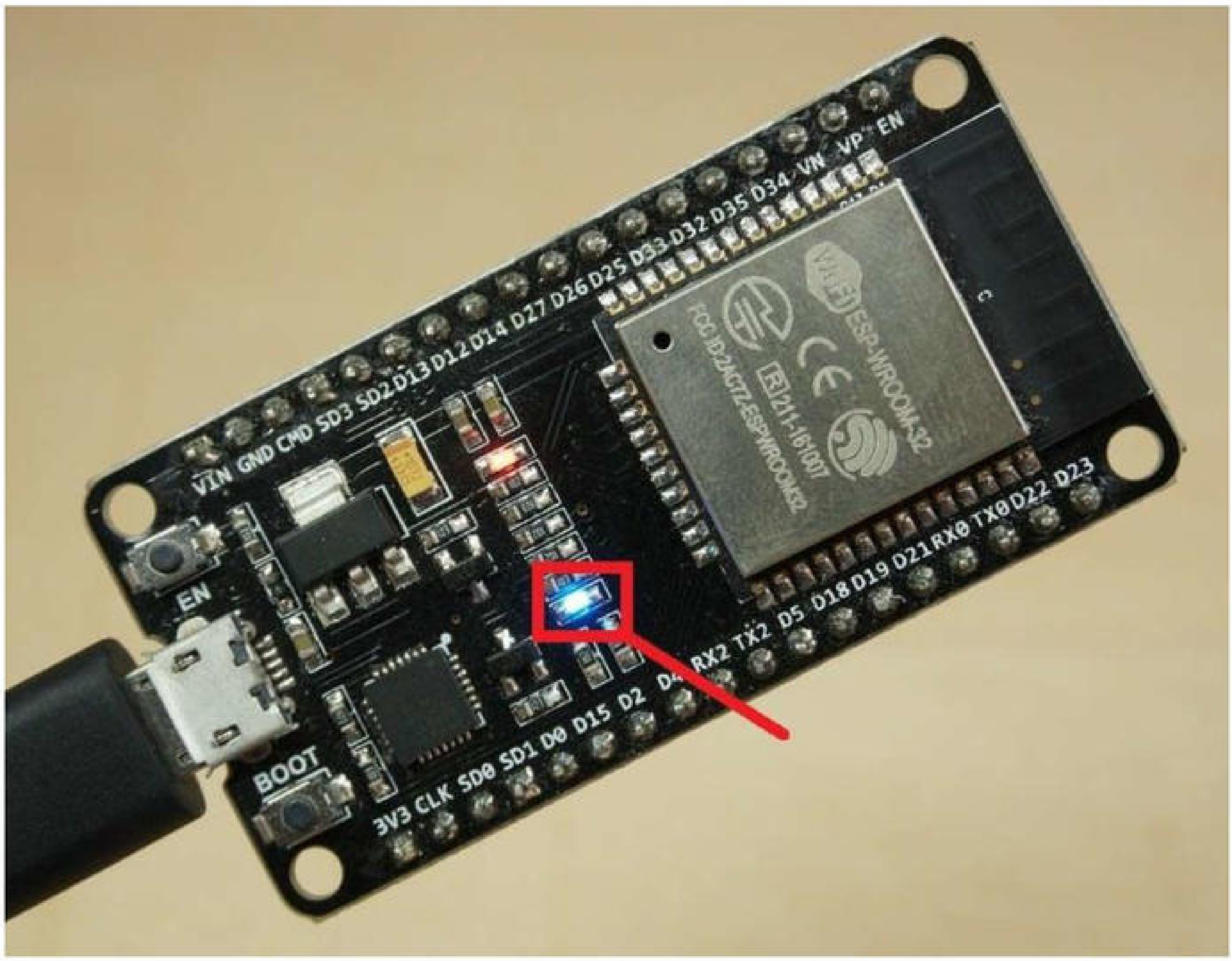


If you're using an ESP32, your Terminal messages should look something as shown in the following figure after pressing the **EN/RST** button:

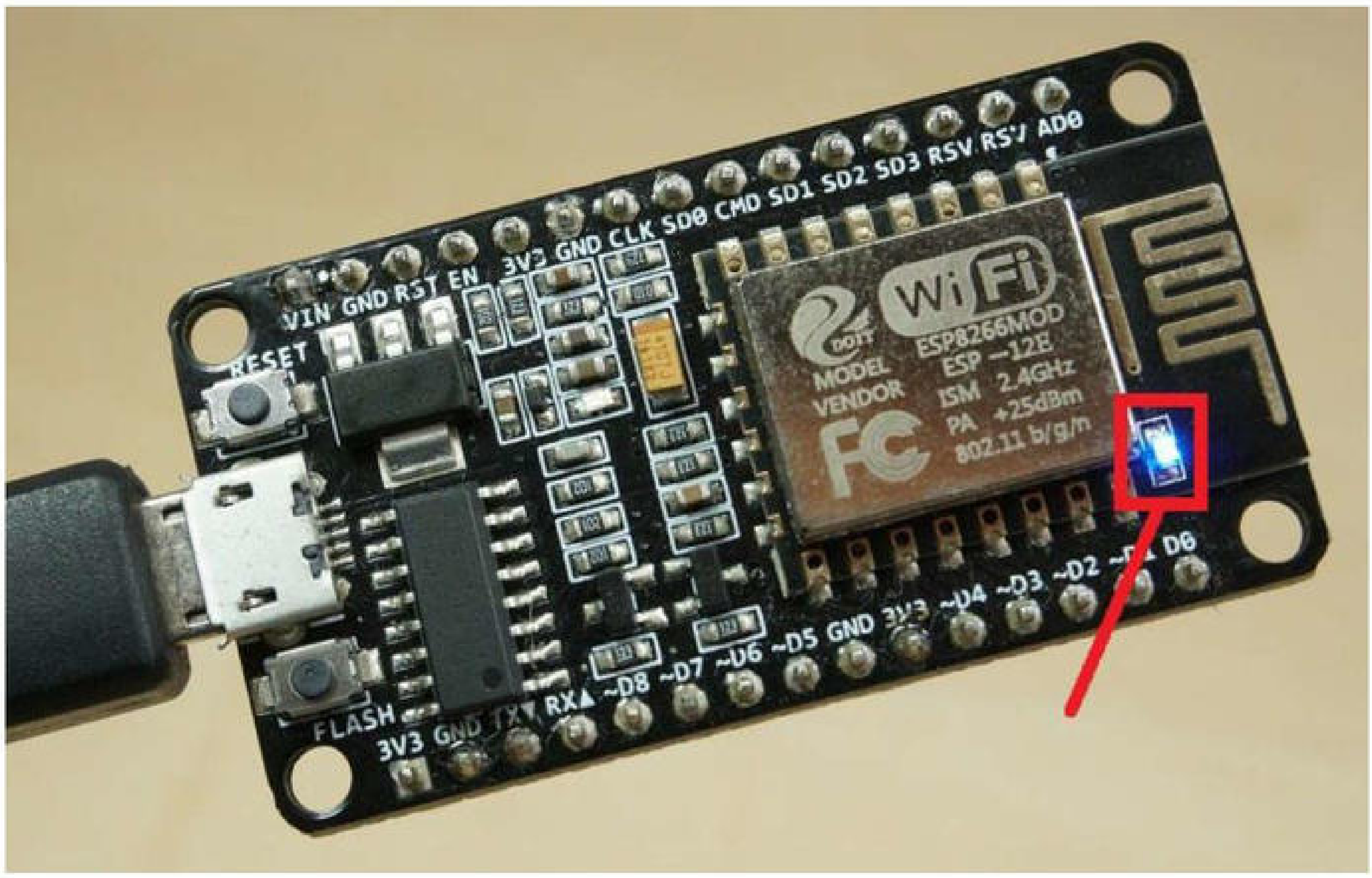


Your ESP32 or ESP8266 on-board LED should be blinking every 500 milliseconds.

Here's where the ESP32's on-board LED is located:



Here's the ESP8266 on-board LED:



Troubleshooting Tips

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

Error #1: *You get the following message:*

```
>>>
Select Serial Port could not open port 'COM4': FileNotFoundError(2, 'The system cannot find the file specified.', None, 2)
```

Or an equivalent message:

```
>>>
could not open port 'COM4': PermissionError(13, 'A device attached to the system is not functioning.', None, 31)
```

Unplug, and plug back your ESP board. Then, double-check that you've selected the right serial port in the **Tools ▸ Serial** menu. Then, click the "**Connect/disconnect**" button to establish a serial communication. You should now be able to upload a new script or re-run new code.

This error might also mean that you have your serial port being used in another program (like a serial terminal or in the Arduino IDE). Double-check that you've closed all the programs that might be establishing a serial communication with your ESP board. Then, unplug and plug back your ESP board. Finally, restart the uPyCraft IDE – try to select the serial port in the **Tools ▸ Serial** menu.

Error #2: *Trouble uploading a new script.*

```
>>>
already in download mode, please wait.
```

Press the "**Stop**" button in uPyCraft IDE (1 or 2 times) to make sure any code that was running stops. After that, press the "**Download and run**" button to upload the new script to your ESP board.

Error #3: *After uploading a new script, if you see the following message*:

```
>>>
Ready to download this file,please wait!
...
download ok
os.listdir('.')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'os' isn't defined
```

Or this message:

```
>>>
Ready to download this file,please wait!
...
download ok
os.listdir('.')
OSError: [Errno 98]
```

It means the new file was uploaded to your board successfully. You can notice that it printed the "**download ok**" message. Press the ESP on-board "EN/RST" button to restart your board and re-run the new uploaded script from the beginning.

Error #4: *Problem restarting your board, running a script or opening the serial port*:

```
>>>
Brownout detector was triggered
```

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

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

Solution: try a different shorter USB cable (with data wires), try a different computer USB port or use a USB hub with an external power supply.

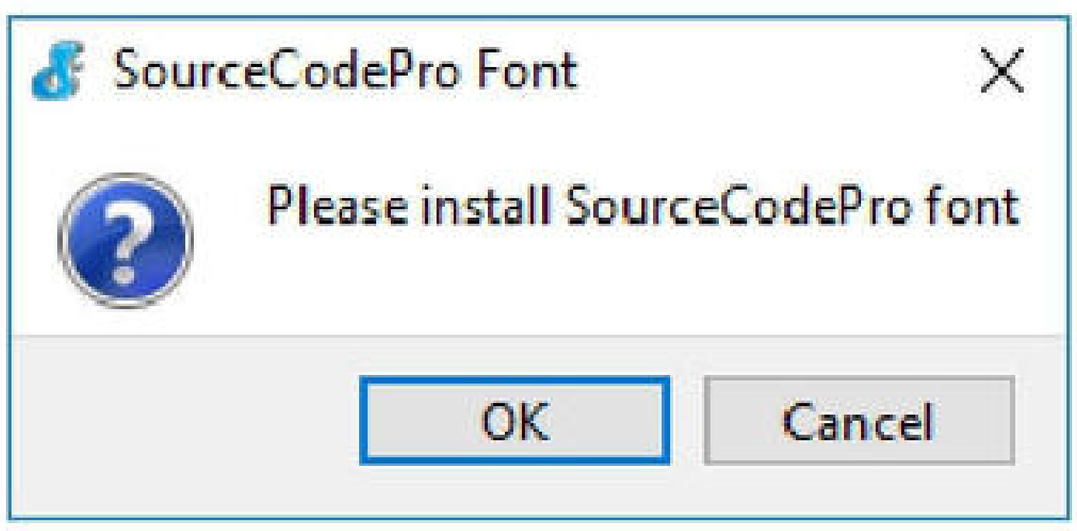
IMPORTANT: if you keep having constant problems or weird error messages, we recommend re-flashing your ESP board with the latest version of MicroPython firmware.

Error #5: *When I try to open a serial communication with the ESP32/ESP8266 in uPyCraft IDE, sometimes it prompts the "Burn Firmware" window asking to re-flash the MicroPython firmware.*

Basically, we think this is what's happening: when you're running a script in your board, sometimes it's busy running that script and performing the tasks. So, you need to try opening the COM port multiple times or restart the ESP to catch it available to establish the serial communication with uPyCraft IDE.

If you're running a script that uses Wi-Fi, deep sleep, or it's doing multiple tasks, I recommend trying 3 or 4 times to establish the communication. If you can't, I recommend re-flash the ESP with MicroPython firmware.

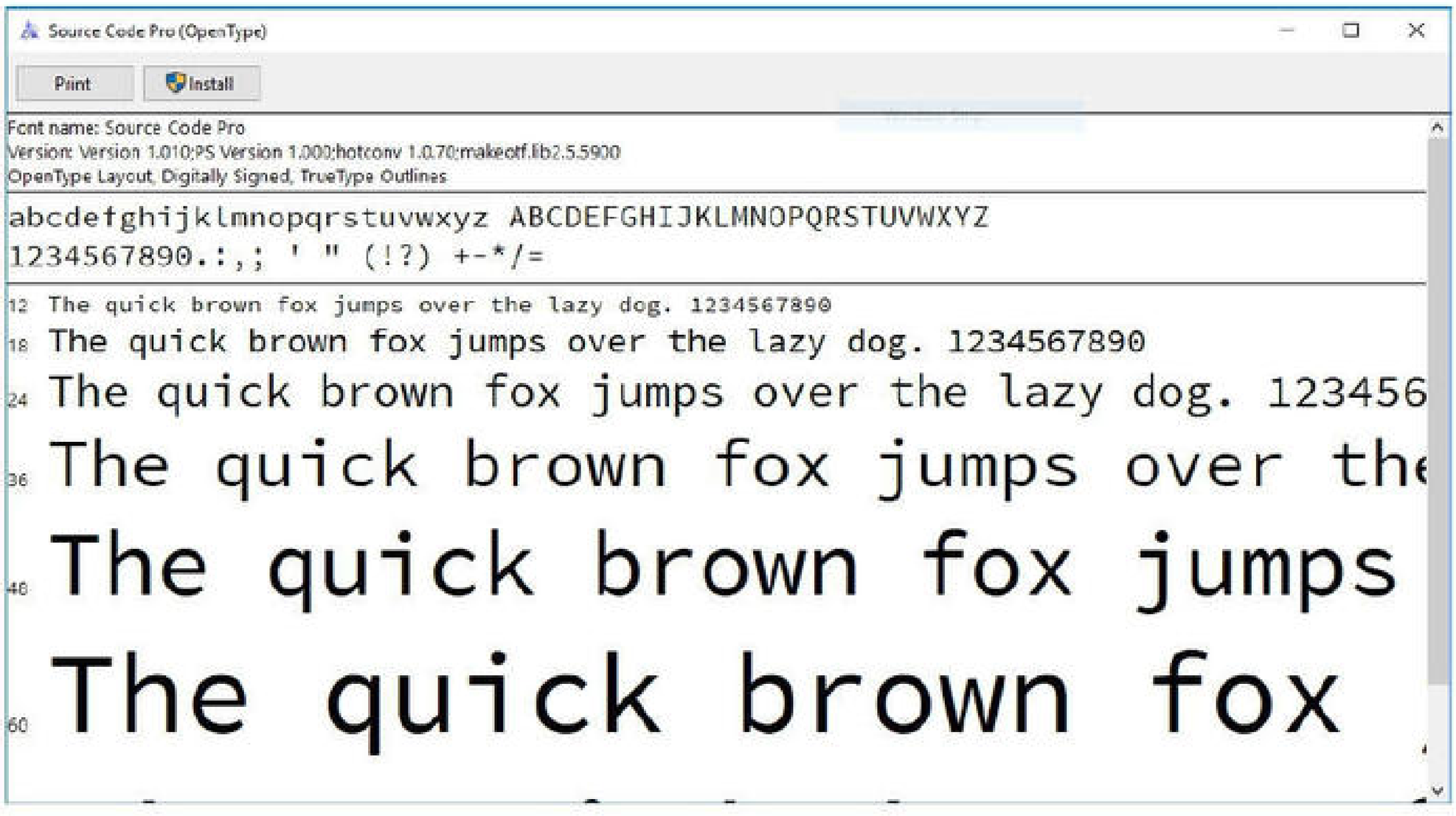
Error #6: *You get the following message: "Please install SourceCodePro font".*



Some computers don't have the SourceCodePro font installed. You can ignore that message and it will still work just fine.

If you want to install the SourceCodePro font, please follow these instructions:

1. Download the following .zip folder and unzip it to find the font file inside: https://randomnerdtutorials.com/SourceCodeProFont
2. Then, you need double-click that file to open it and press the "Install" button:



After that, re-open uPyCraft IDE and it will not prompt that warning message again.

Error #7: *You get one of the following indentation error messages:*

```
IndentationError: unexpected indent
```

Or

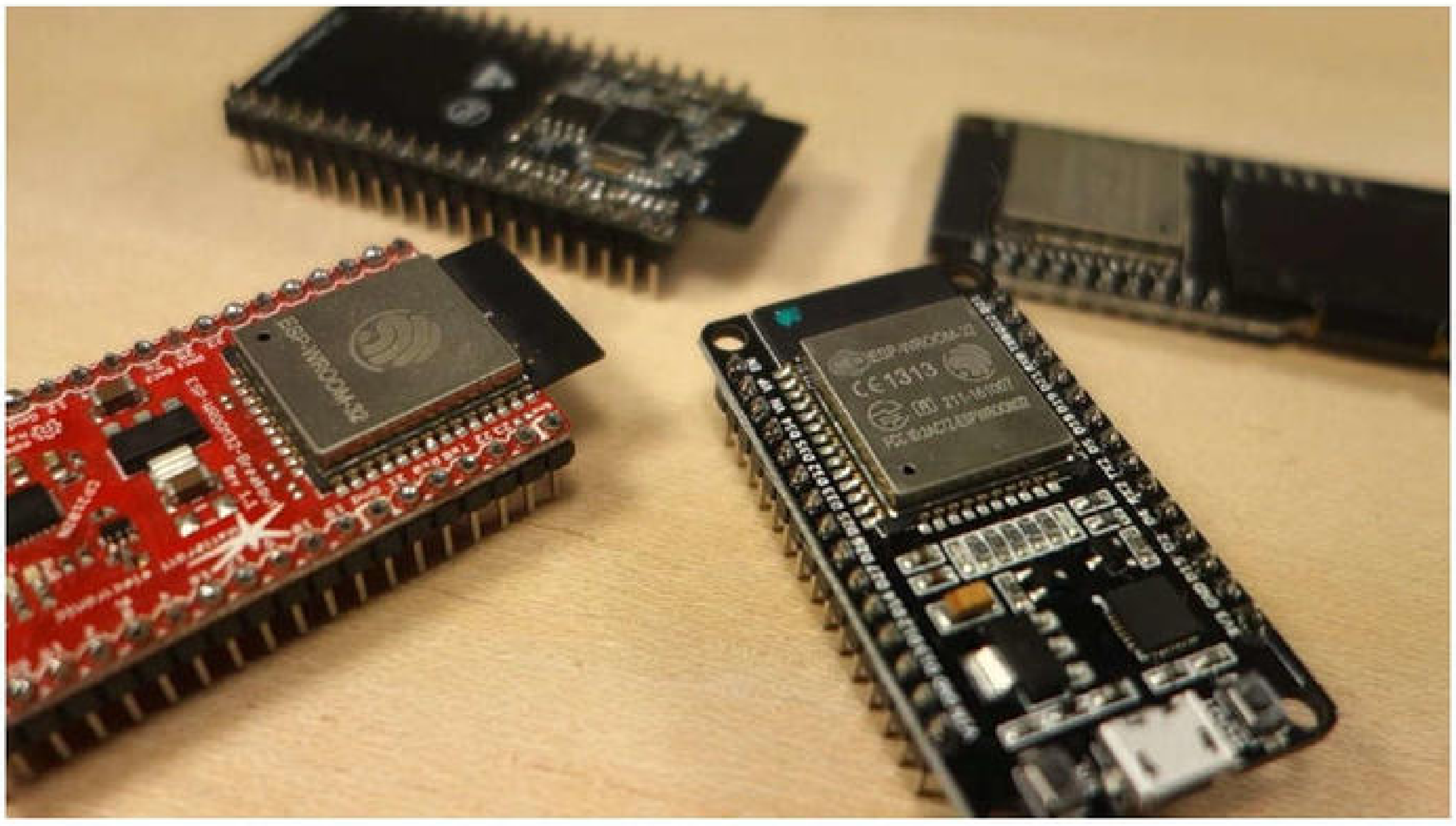
```
IndentationError: unindent doesn't match any outer indent level
```

Or

```
SyntaxError: invalid syntax
```

If you upload a new script to your ESP board and it prompts one of the preceding error messages, it means that there's a space or multiple spaces missing in your script. Python doesn't use *;* or *{}* to know where each line ends. Python uses indentation (everything is done using spaces). Double-check that all the spaces are properly inserted in your script. To avoid indentation errors, we recommend copying/pasting the code directly from the GitHub link provided in each example.

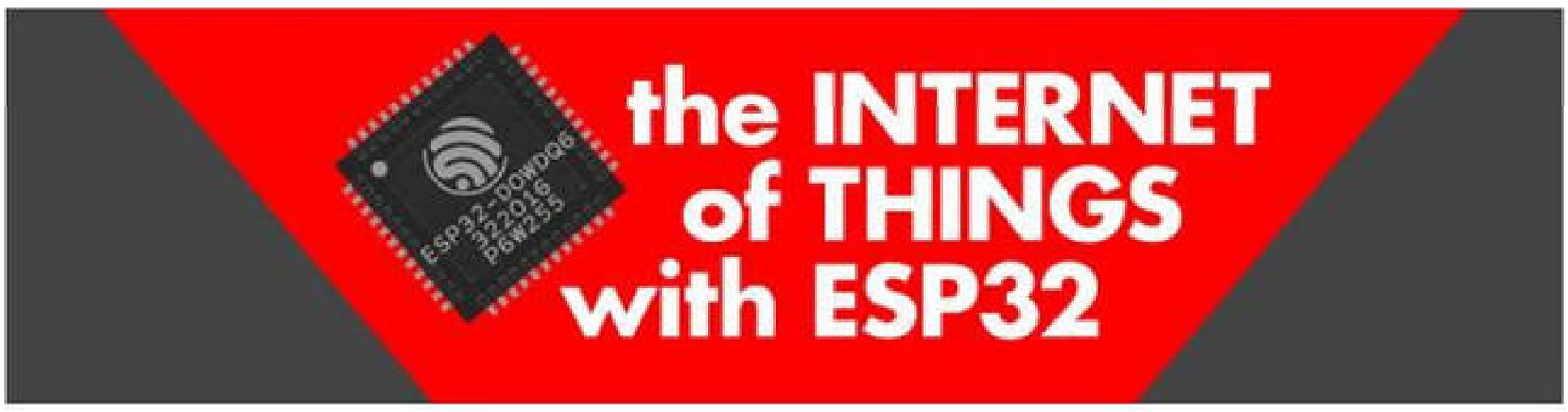
Introducing the ESP32 Board



This section is an introduction to the ESP32 board, which is the ESP8266 successor.

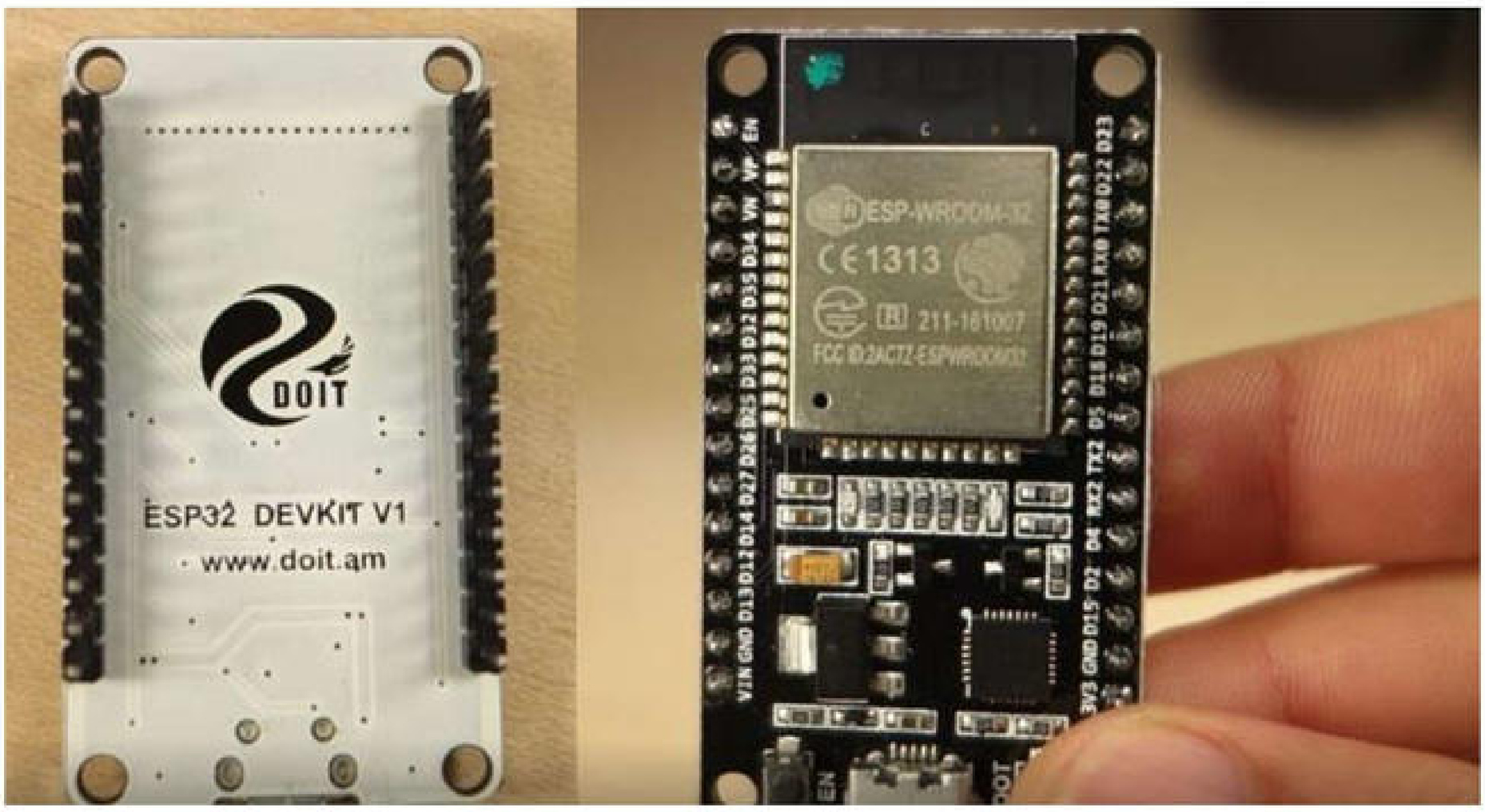
The ESP32 is loaded with lots of new features. It combines Wi-Fi and Bluetooth wireless capabilities.

There are many ESP32 development boards. I encourage you to visit the ESP32.net website where each ESP32 chip and development board is listed. You can compare their differences and features.



Introducing the ESP32 DOIT DEVKIT V1 Board

Throughout this course, we'll be using the ESP32 DEVKIT V1 DOIT board for the ESP32 examples and projects, but any other ESP32 with the ESP-WROOM-32 chip will work just fine.



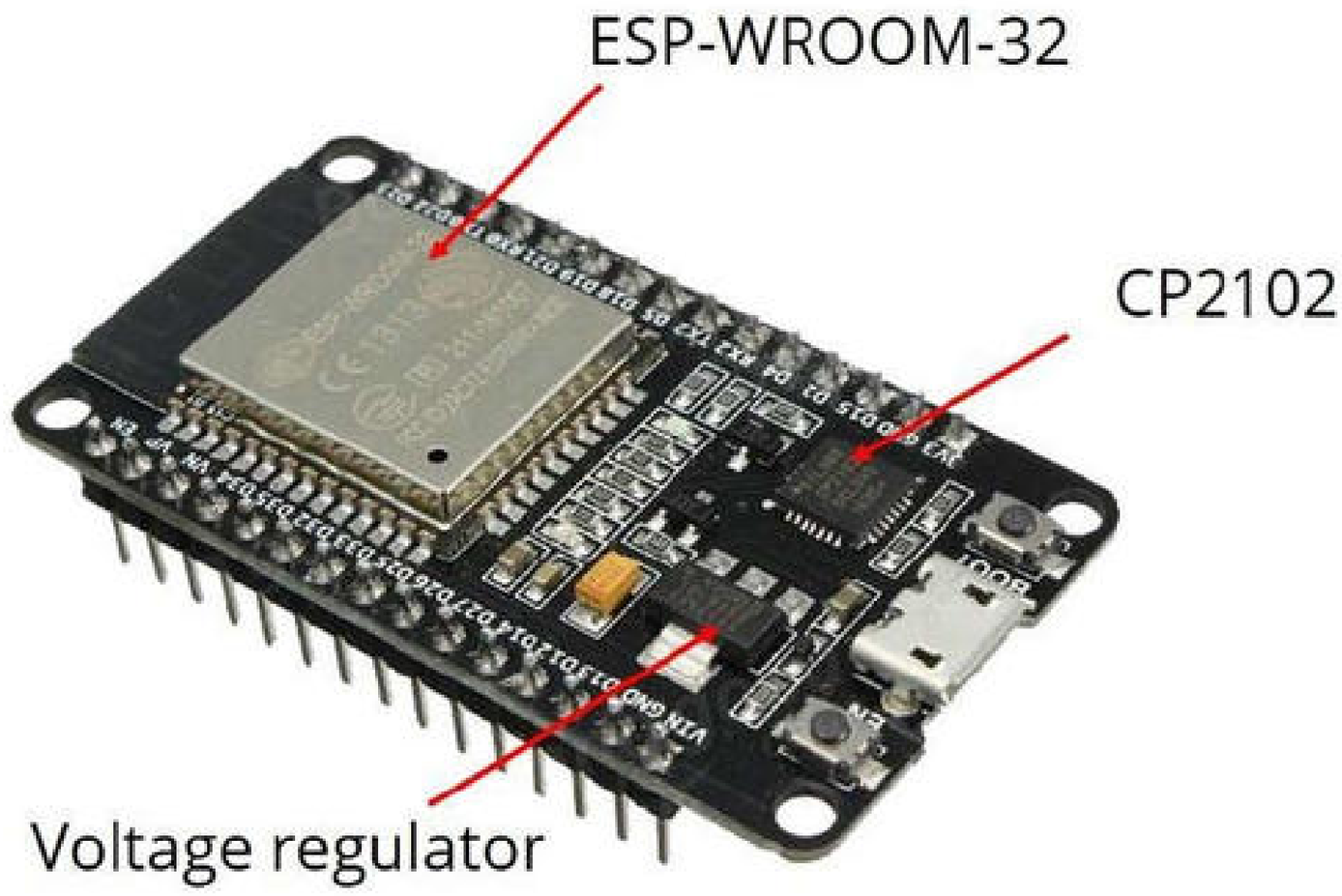
Here's just a few examples of boards that are very similar and compatible with the projects that will be presented throughout this book.

Any ESP32 board should work with this course with small changes in the wiring. For a comparison between the most common ESP32 boards, you can read the following article:

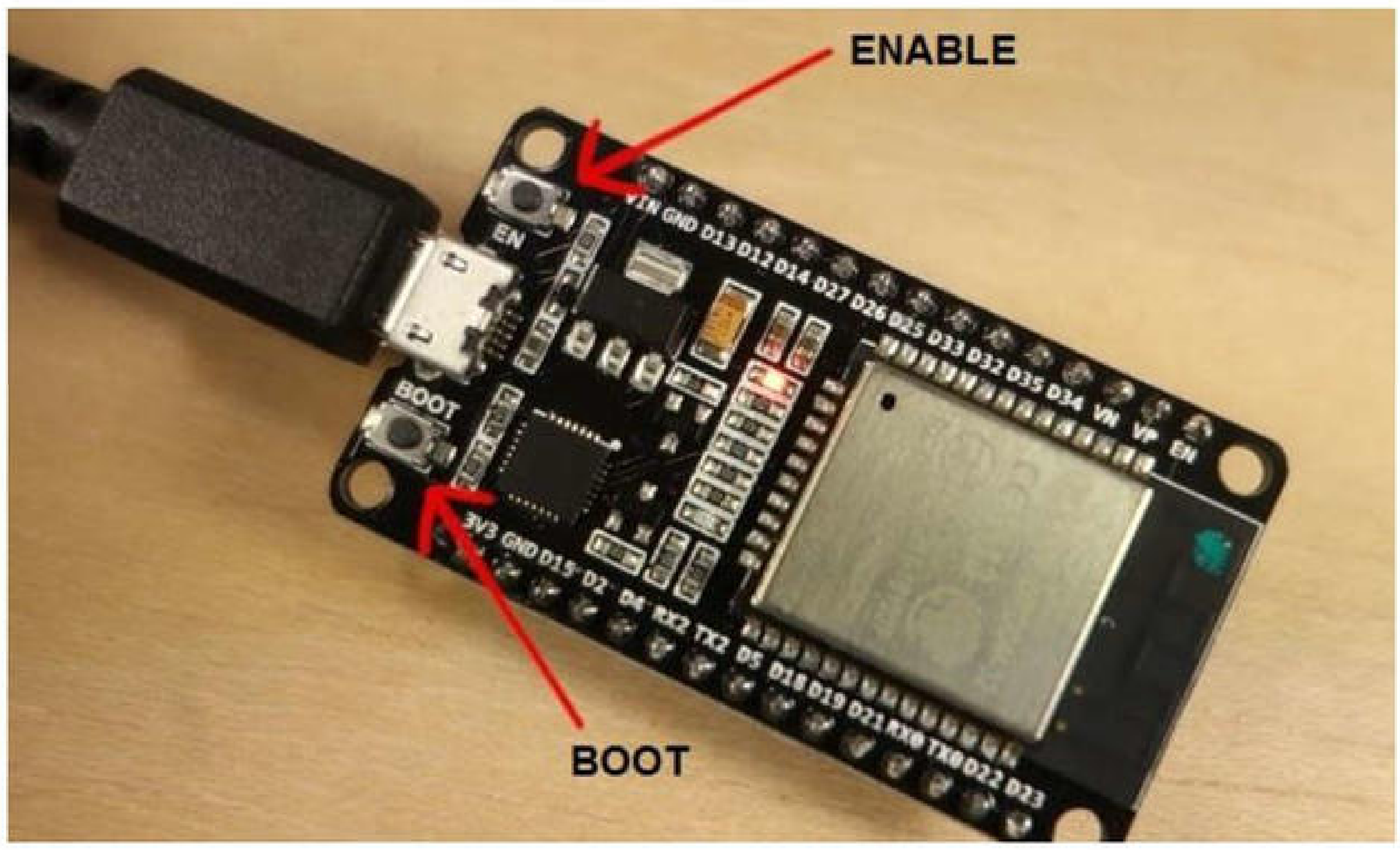
- ESP32 Development Boards Review and Comparison

ESP32 Features

The ESP32 V1 DOIT board comes with the ESP-WROOM-32 chip. It has a 3.3V voltage regulator that drops the input voltage to power the ESP32 chip. It comes with a CP2102 chip that allows you to plug the ESP32 to your computer to program it without the need for an FTDI programmer.



The board has two on-board buttons: the ENABLE and the BOOT button.



If you press the ENABLE button, it reboots your ESP32. If you hold down the BOOT button and then press the enable, the ESP32 reboots in programming mode. You don't need to worry about these details at the moment.

ESP32 Specifications

When it comes to the ESP32 chip specifications, you'll find that:

- The ESP32 is dual core, this means it has 2 processors.
- It has Wi-Fi and bluetooth built-in.
- It runs 32-bit programs.
- The clock frequency can go up to 240 MHz and it has a 512 kB RAM.
- This board (ESP32 V1 DOIT board) comes with 30 pins, 15 in each row. (There's an alternative version of this board with 36 pins.)
- It also has wide variety of peripherals available, like: capacitive touch, ADCs, DACs, UART, SPI, I2C and much more. We'll explore some of these functionalities later in the course.

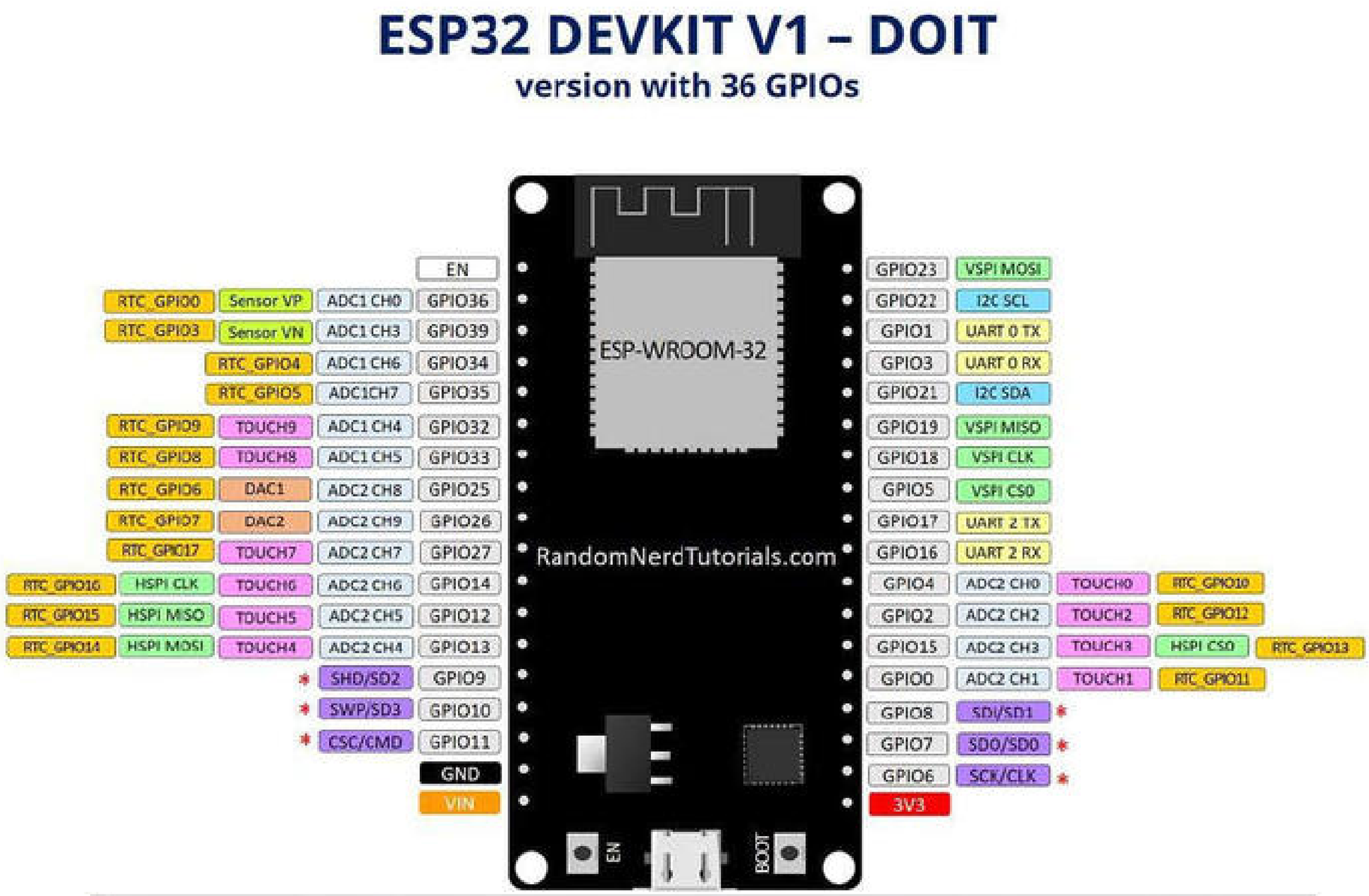
Specifications - ESP32 DEVKIT V1 DOIT	
Number of cores	2 (Dual core)
Wi-Fi	2.4 GHz up to 150 Mbit/s
Bluetooth	BLE (Bluetooth Low Energy) and legacy Bluetooth
Architecture	32 bits
Clock frequency	Up to 240 MHz
RAM	512 KB
Pins	30
Peripherals	Capacitive touch, ADCs (analog-to-digital converter), DACs (digital-to-analog converter), I^2C (Inter-Integrated Circuit), UART (universal asynchronous receiver/transmitter), CAN 2.0 (Controller Area Network), SPI (Serial Peripheral Interface), I^2S (Integrated Inter-IC Sound), RMII (Reduced Media-Independent Interface), PWM (pulse width modulation), and more.

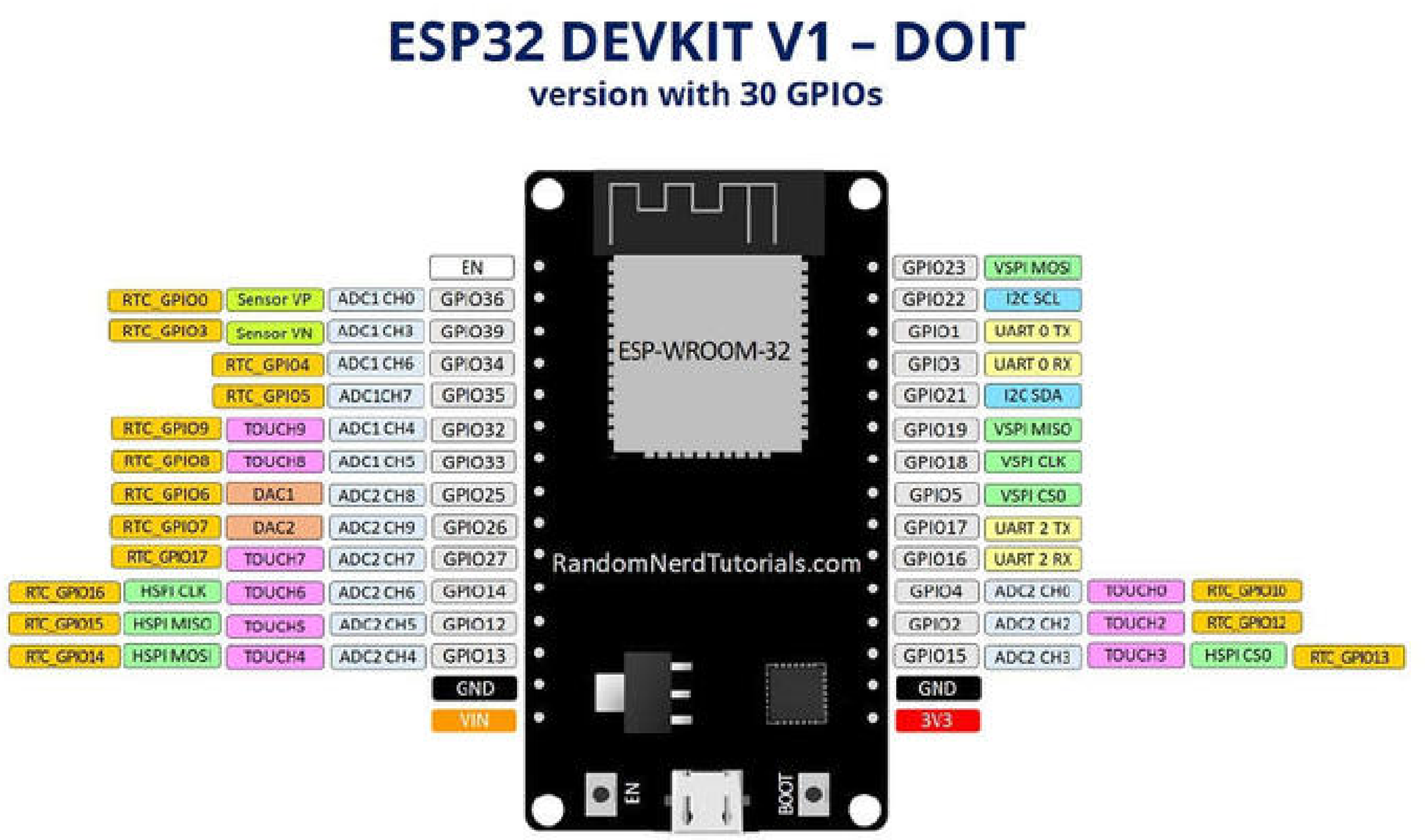
For more information about the ESP32 GPIOs, we recommend reading the following article:

- ESP32 Pinout Reference: Which GPIO pins should you use?

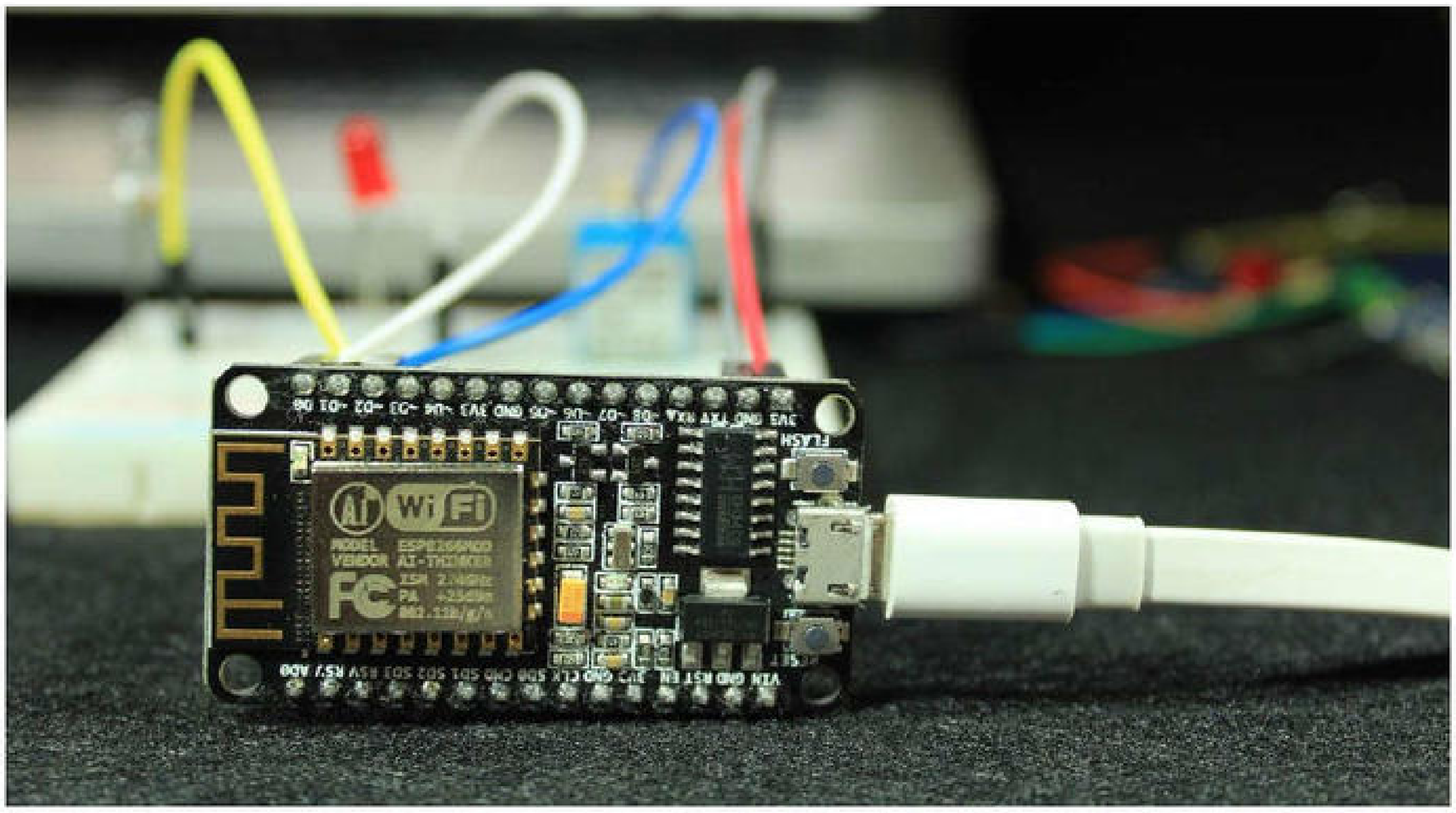
ESP32 Pinout

The following figures clearly describes the board GPIOs and their functionalities. The DEVKIT V1 DOIT board is available in two different versions: with 30 and with 36 GPIOs.





Introducing the ESP8266 Board



The ESP8266 is a Wi-Fi module great for IoT and Home Automation projects. This section is a quick getting started guide for the ESP8266 development board. If you're already familiar with the ESP8266 you can skip this section.

About the ESP8266

The ESP8266 is a $4 (up to $10) Wi-Fi module. It allows you to control inputs and outputs as you would do with an Arduino, but it comes with Wi-Fi. So, it is great for home automation/internet of things applications. So what can you do with this low cost module?

- create a web server
- send HTTP requests
- control outputs
- read inputs and interrupts
- send emails
- post tweets
- build home automation and IoT projects

ESP8266 specifications

- 11 b/g/n protocol
- Wi-Fi Direct (P2P), soft-AP
- Integrated TCP/IP protocol stack
- Built-in low-power 32-bit CPU
- SDIO 2.0, SPI, UART

Comparing the ESP8266 with other Wi-Fi solutions on the market, it is a great option for most "Internet of Things" projects! It's easy to see why it's so popular: it only costs a few dollars and can be integrated in advanced projects. We've published dozens of free ESP8266 projects and tutorials.

ESP8266 Versions

The ESP8266 comes in a wide variety of versions (as shown in the figure below). The ESP-12E or often called ESP-12E NodeMCU Kit is currently the most practical version, in our opinion.

For the ESP8266 examples and projects in this course, we used the ESP8266-12E NodeMCU Kit, the one that has built-in programmer. The built-in programmer makes it easy to prototype and upload your programs.

For a comparison between the different ESP8266 boards, you can read the following article:

- Best ESP8266 Wi-Fi Development Boards

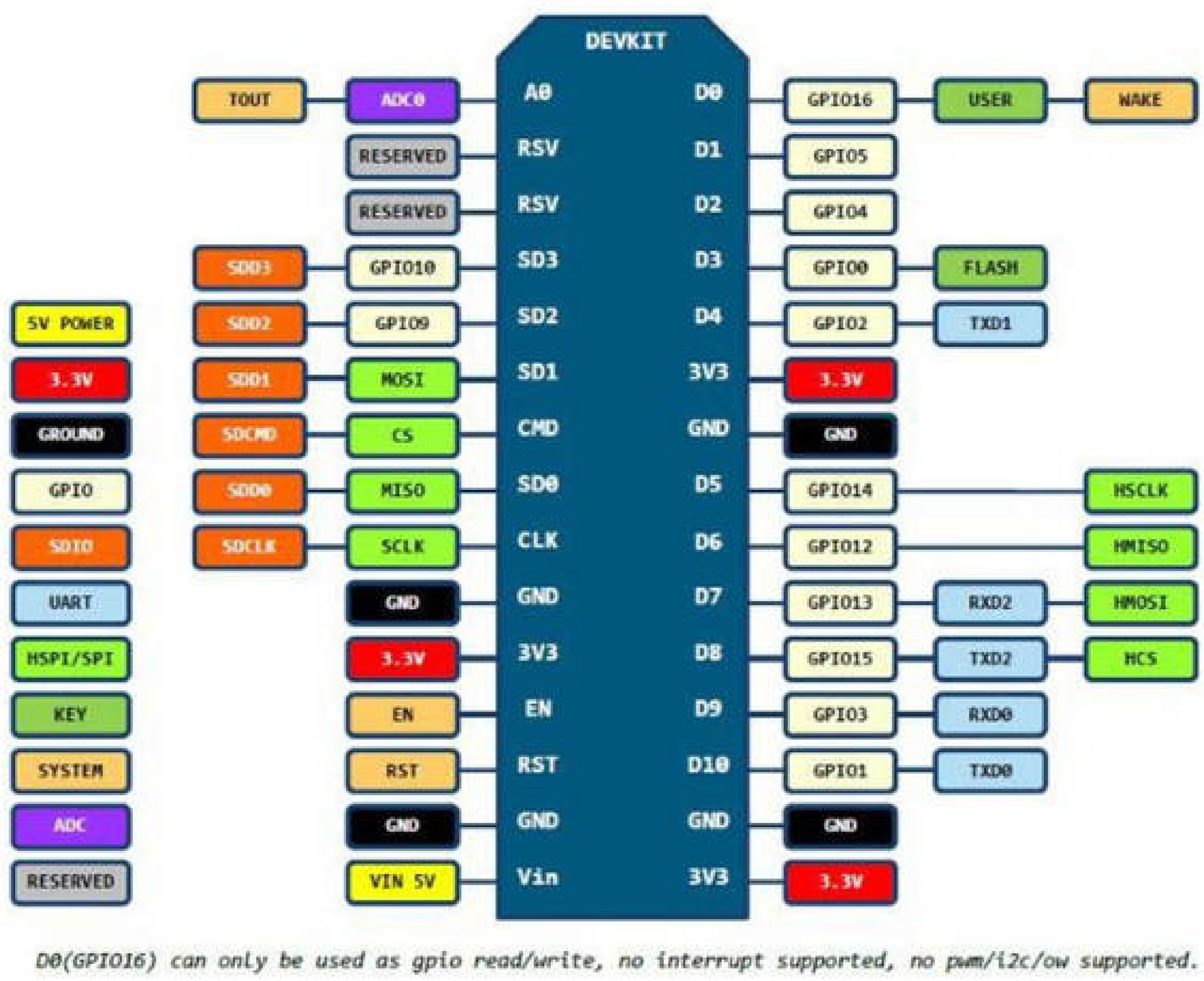
The most widely used ESP8266 development boards are the ESP-01, the ESP-12E and the WeMos D1 Mini.

Note: we don't recommend using an ESP-01 to follow this course because most versions don't have enough memory to support the full MicroPython firmware. There are builds of the ESP8266 firmware tailored for modules with only 512kbytes of flash. However, certain features are disabled to get the firmware down to this size (see the MicroPython downloads webpage). Please note that this won't be covered on the eBook.

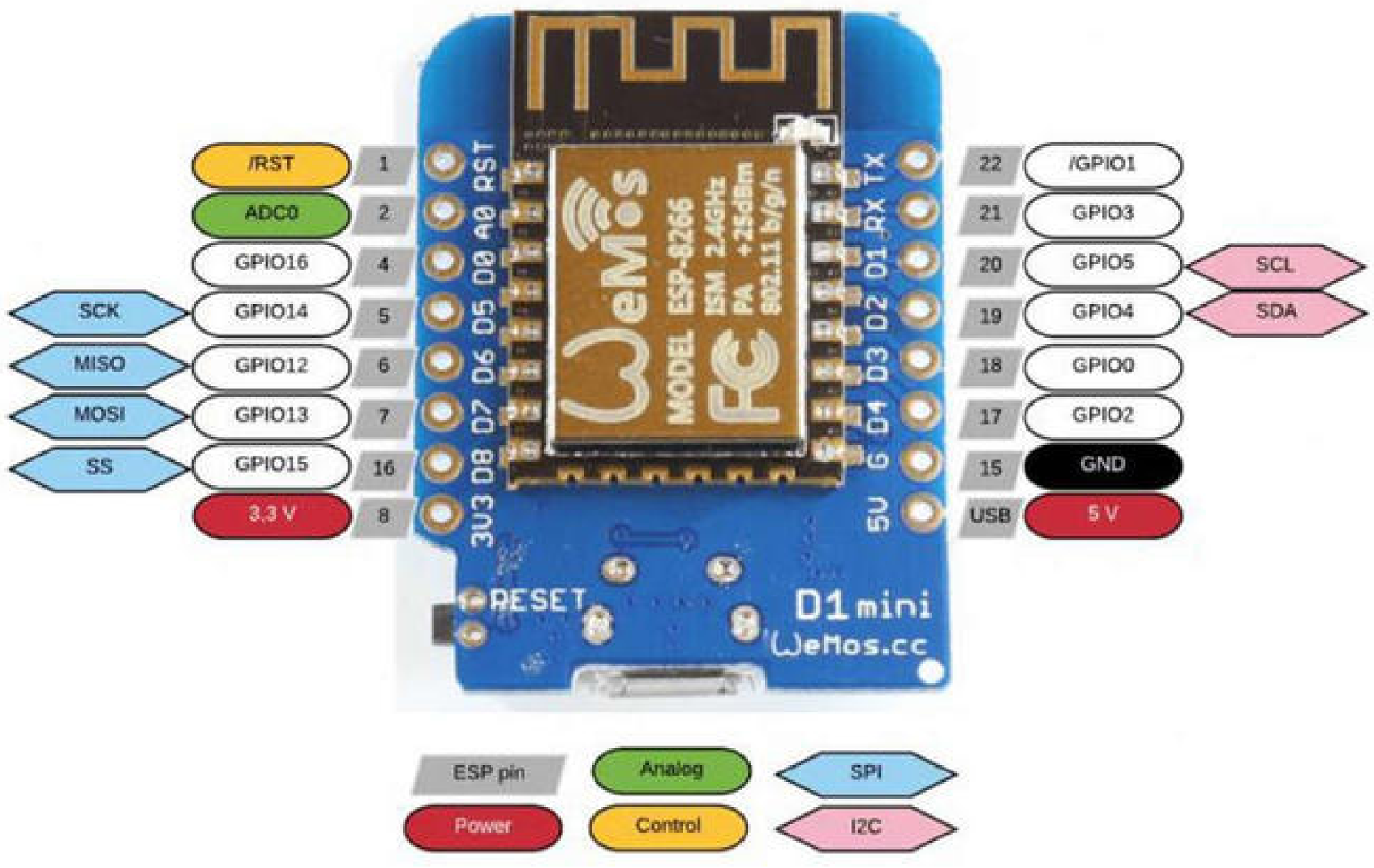
ESP8266 Pinout

We'll show you the pinout for the ESP-12E NodeMCU Kit and Wemos D1 mini. If you're using another development board, make sure you search for the right pinout.

ESP-12E NodeMCU Kit Pinout



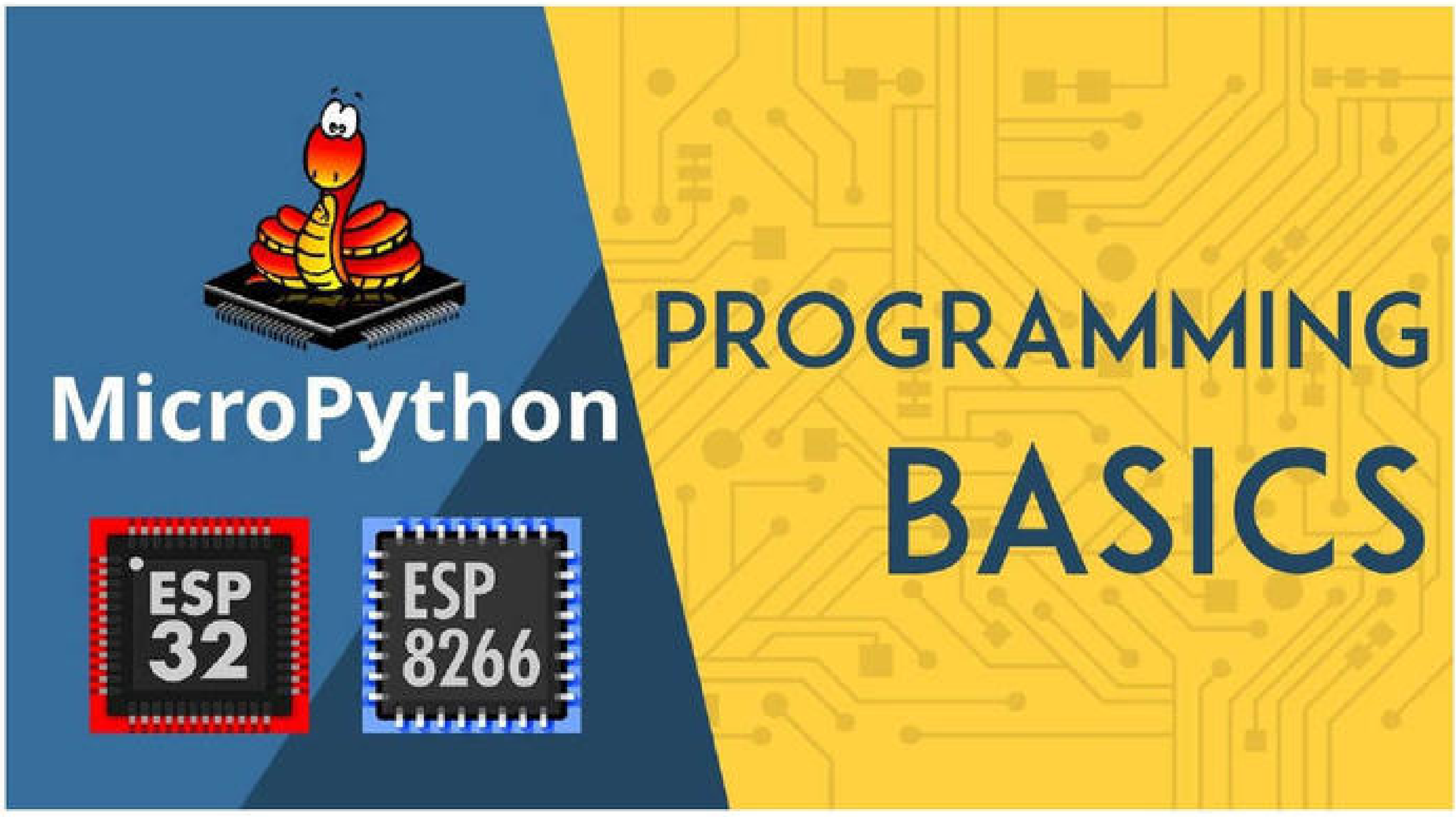
Wemos D1 Mini Pinout



MODULE 2

Python/MicroPython Programming Basics

MicroPython Programming Basics



MicroPython is a re-implementation of Python programming language targeted for microcontrollers and embedded systems like the ESP32 or ESP8266.

Programming in MicroPython is very similar to programming in Python: all language features of Python are also in MicroPython, apart from a few exceptions. Because microcontrollers and embedded systems are much more limited than our computers, MicroPython does not come with the full standard library by default.

If you already know how to program in Python, programming in MicroPython is the same. You just need to bear in mind that MicroPython is used for constrained devices. Therefore, you must keep your code as simple as possible.

This section explains the basics of Python programming language syntax that also apply to MicroPython, like:

- Mathematical operators
- Relational operators
- Data types
- print() function
- Comments

- Conditional statements
- While and for loops
- User defined functions
- Classes and objects
- Modules

Mathematical Operators

MicroPython can perform mathematical operations. The following table shows the mathematical operators supported:

Operator	Mathematical Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
//	Division, discarding the decimal point
%	Remainder after division

In the Shell, try several operations to see how it works. For example:

```
>>> 2+2*9-3
17
>>> 28594/2312
12.36765
>>> 214522236/7.5
2.860297e+07
>>> 23//2
11
>>> 25%3
1
```

You can perform other mathematical operations if you import the `math` module, like square root, trigonometric functions, logarithm, exponentiation, etc.

Relational Operators

You can make comparisons using relational operators. These compare the values on either sides and show the relation between them.

Operator	Description
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Try several comparisons and test the result:

```
>>> 2 == 3
False
>>> 4 == 4
True
>>> 3 > 2
True
>>> 489808234 != 2223
True
>>> 4.5 >= 4.5
True
```

Assigning Values to Variables

In Python, you don't need to declare what type each variable is. If you're used to program your boards using Arduino IDE, you know that you need to declare the type of a variable when creating a new variable. There isn't such thing in Python.

Variables are simply a storage placeholder for values: number or text. To assign a value to a variable you use the equal sign (=), with the variable name on the left and the value on the right.

For example, to create a variable to hold the GPIO number where an LED is connected to, you can simply type the following:

```
led_pin = 23
```

In the Arduino IDE, you would have something like:

```
const int led_pin = 23;
```

As you can see, Python is much simpler than programming in C (in Arduino IDE).

Note: the names you give variables can't have spaces and are case sensitive, so `led_pin` is different from `LED_PIN` or `Led_Pin`.

Data Types

Variables can store several types of values, not just whole numbers. That's where data types come in. A *data type* is a classification of a value that tells what operations can be done with the value and how it should be stored.

The following table shows the data types we'll use more often in our projects.

Data type	Description
int (Int)	Integer (whole number)
float (Float)	Number with a decimal point
str (String)	Set of characters between quotation marks
bool (Boolean)	True or False

Let's create variables with different data types:

```
>>> a = 6
>>> b = 95.32
>>> c = 'Hello World!'
>>> d = True
```

- The first value assigned to `a`, is an **integer**, which is a whole number.
- The `b` variable contains a **float** value, which is a number with a decimal.
- The third value, `'Hello World!'`, is a **string**, which is a series of characters. A string must be put inside single `'Hello World!'` or double quotation `"Hello World!"` marks.
- Finally, `d` is a **Boolean**, which is a type that can only take either `True` or `False`.

There is a function to check the data type of a variable: the `type()` function. This function accepts as argument the variable you want to check the data type.

```
type(variable)
```

For example, after declaring the variables in the previous example `a`, `b`, `c`, and `d`, you can check its data type. For example:

```
>>> type(a)
```

Returns:

```
<class 'int'>
```

This tells that `a` is an int (integer). Experiment with the other variables and you should get:

```
>>> type(b)
<class 'float'>
>>> type(c)
<class 'str'>
>>> type(d)
<class 'bool'>
```

print() Function

The `print()` function prints the message between parentheses into the Shell. This is useful to debug the code and keep track of what's going on. For example:

```
>>> print('LED is on')
LED is on
```

Comments

Comments in Python start with the hash character # and continue to the end of the line. A comment is useful to add "notes" in your program or to tell anyone who reads the program what the script does. This doesn't add any functionality to your program. For example:

```
# This is just a comment
```

Because in MicroPython we are working under constrained conditions, there are occasions in which you should avoid adding comments to save space on the flash memory.

Conditional Statements

To write useful programs, you'll probably need to perform different actions depending on whether a certain condition is True or False. We're talking about conditional statements. They have the following structure:

```
if <expr1>:
  <statement1>
elif <expr2>:
  <statement2>
elif <expr3>:
  <statement3>
(...)
else:
  <statementn>
```

`<expr>` is a Boolean expression and it can be either `True` or `False`. If it is `True`, the `<statement>` right after it is executed. The `<statement>` should be indented so that Python knows what statement belongs to each expression.

The `elif` statement stands for `else if` and runs only if the first `if` condition is not `True`.

The `else` statement only runs if none of the other expressions are True.

There's no limit to the number of `elif` statements in a program. It's also not necessary to include an `else` clause, but if there is one, it must come at the end.

In the Arduino IDE, we use `{}` curly brackets to define code blocks. With MicroPython, we use indentation. Additionally, you need to use a colon **:** after each expression. Contrary to the Arduino IDE, the expression doesn't need to be inside parentheses.

IMPORTANT: Python's standard indentation is 4 spaces. In MicroPython, indentation should be only 2 spaces to fit more code into the flash memory.

While and For loops

Loops allow you to execute a block of code multiple times for as long as a condition is met. There are two kinds of loops: while and for loops.

For example, you can print all numbers from 1 to 10 with a while loop:

```
number = 1
while number <= 10:
  print(number)
  number = number + 1
```

The code that belongs to the `while` loop, indicated by the indentation, is executed as long as the value in the variable `number` is less than or equal to (<=) 10. In every loop, the current number is printed and then we add 1 to it.

You can also print numbers from 1 to 10 using a for loop, like this:

```
number = 1
for number in range(1, 11):
  print(number)
```

The `for` loop is executed as long as the value in the `number` variable is within the range of 1 and 11. The `range()` function automatically assigns the next value to the `number` variable, until 1 below the final number you specify.

You should use a `for` loop when you want to repeat a block of code a certain number of times. Use a while loop when you want to repeat code until a certain condition is no longer met. In some situations, you can use either one, oftentimes one is more suitable than the other.

Similar to the conditional statements, the for and while Boolean expressions should have a colon **:** right after them, and the expressions to be executed should be indented.

User-defined Functions

To define a new function, you use the word `def` followed by the name you want to give the function and a set of parentheses (and arguments inside, if necessary). After the parentheses you add a colon : and then tell the function what instructions to perform. The statements should be indented with 2 spaces (in MicroPython):

```
def my_function(<arg1>, <arg2>, ...):
  <statement>
  (...)
  return
```

For example, a function that converts the temperature in Celsius to Fahrenheit could be as follows:

```
def celsius_to_fahrenheit(temp_celsius):
  temp_fahrenheit = temp_celsius * (9/5) + 32
  return temp_fahrenheit
```

The `celsius_to_fahrenheit()` function accepts as argument a temperature in Celsius (`temp_celsius`). Then, it does the calculation to convert the temperature. Finally, it returns the temperature in Fahrenheit (`temp_fahrenheit`).

Note: functions don't necessarily need to return a variable. They could just perform some work without the need to return anything.

Classes and Objects

Python is an object-oriented programming language (OOP). There are two important concepts you need to understand about OOP: *classes* and *objects*.

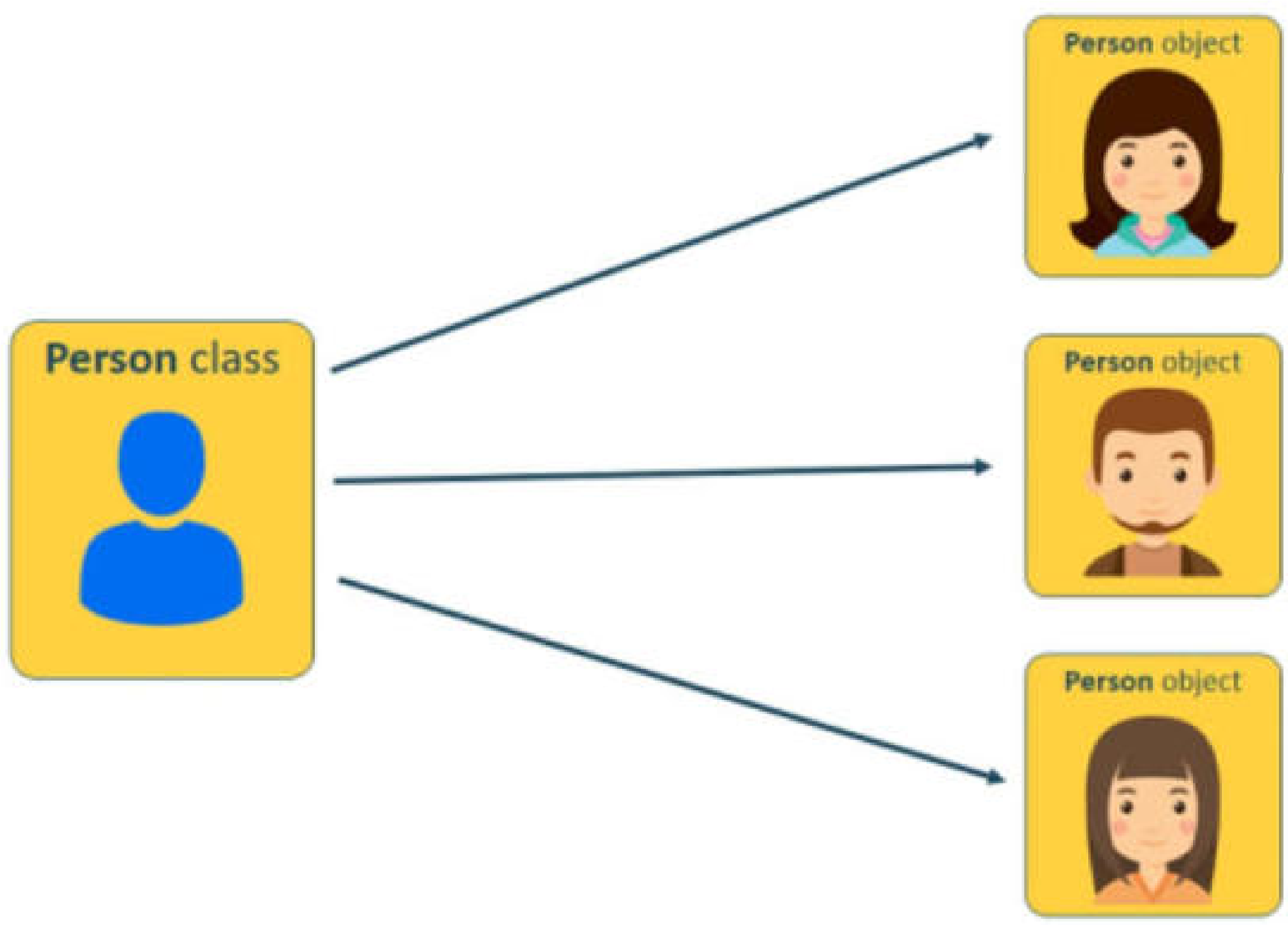
A **class** is a blueprint for objects. It defines a set of attributes (data and functions) that characterize an object. The functions inside of a class are called *methods*. Classes are defined by the keyword `class` followed by the name of the class. For example:

```
class MyClass:
  (...)
```

Note: by convention, classes' names in Python should be CapWords. However, you can give whatever name you want.

An **object** is an instance of a class. It's simply a collection of data and methods into a single entity. Through the object, you can use all functionalities of its class. Confused? Let's look at a simple example.

If we would like to define several persons in a Python program using the same attributes, we can think of the term person as a class. We may want to define a person using attributes like name, age, country, etc.



So, we can create a class called `Person`. Our class will have the following attributes: `name`, `age`, and `country`. You can add as many attributes as you want. We'll also create a function (method) that prints a description of the person based on its attributes:

```
class Person:
  name = ""
  age = 0
  country = ""
  def description(self):
    print("%s is %d years old and he is from %s." %(self.name, self.age, self.country))
```

As you can see, we define a new class by using the keyword `class`, followed by the name we want to give to the class: `Person`.

Inside the `Person` class, we define several variables to hold values. By default, the `name` and `country` are empty strings, and the `age` is 0. Then, we also define a function (method) that prints all the variables values into the Shell.

All functions inside a class should have the `self` parameter as argument and other arguments, if needed.

The `self` parameter refers to the object itself. It is used to access variables that belong to the class. For example, to access the `name` variable inside the class, we should use `self.name`.

Now that we've create a class, we can create as many `Person` objects as we need using that class. The `Person` object will have a `name`, `age`, and `country`. We'll also be able to print its description using the `description()` method we've created.

For example, to create a new `Person` object called `person1`:

```
>>> person1 = Person()
```

Set the object's properties

To set the `name`, `age`, and `country` of the `person1` object. You can do it as follows:

```
>>> person1.name = "Rui"
>>> person1.age = 25
>>> person1.country = "Portugal"
```

Calling methods

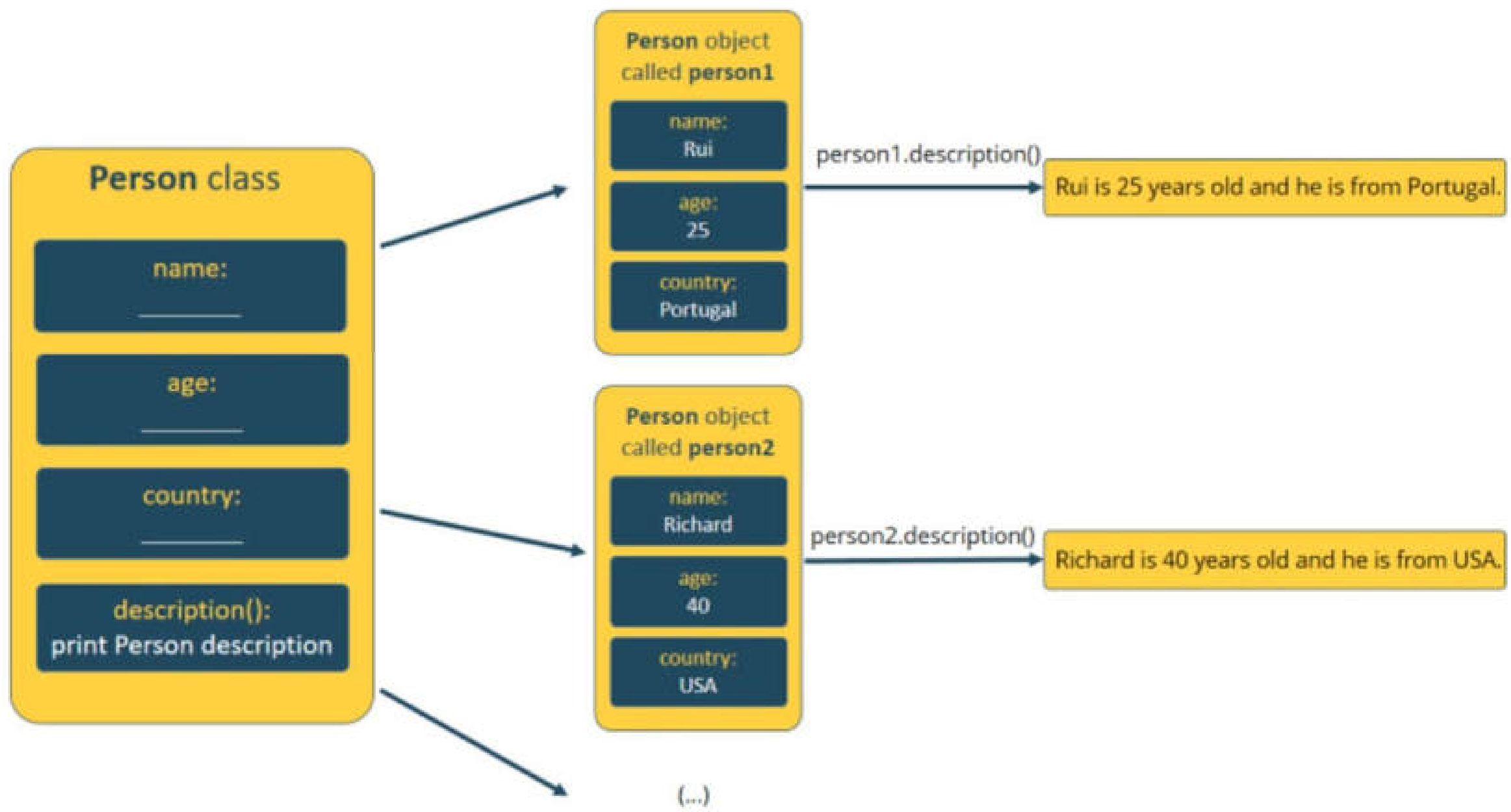
Later in your code, you can use the created `description()` method on any `Person` object. To call the `description()` method on the `person1` object:

```
>>> person1.description()
```

This should print the following:

```
Rui is 25 years old and he is from Portugal.
```

You should now understand that you can create as many objects as you want using the same class and you are able to use the available methods with all the objects of that class.



The constructor method

Instead of having to define a class, and then set the object's properties, which can be time consuming, you can use the *constructor method* inside your class.

The constructor method is used to initiate data as soon as an object of a class is instantiated. The constructor method is also known as `__init__` method. Using the `__init__` method, the `Person()` class looks as follows:

```
class Person():
  def __init__(self, name, age, country):
    self.name = name
    self.age = age
    self.country = country
  def description(self):
    print("%s is %d years old and he is from %s." %(self.name, self.age, self.country))
```

Then, to instantiate a `Person` object with the same attributes we've defined earlier, we just need to do the following:

```
>>> person1 = Person("Rui", 25, "Portugal")
```

If you call the `description()` method on the `person1` object, you'll get the same result:

```
>>> person1.description()
Rui is 25 years old and he is from Portugal.
```

Modules

A **module** is a file that contains a set of classes and functions you can use in your code – you can also call it library. To access the classes and functions inside that code, you just need to import that module into your code.

You can create your own modules or use already created modules from the standard Python library. When it comes to MicroPython, it only comes with a small subset of the standard Python library, but it does come with a set of modules to control GPIOs, make networks connections and much more.

Importing modules/libraries is as simple as using:

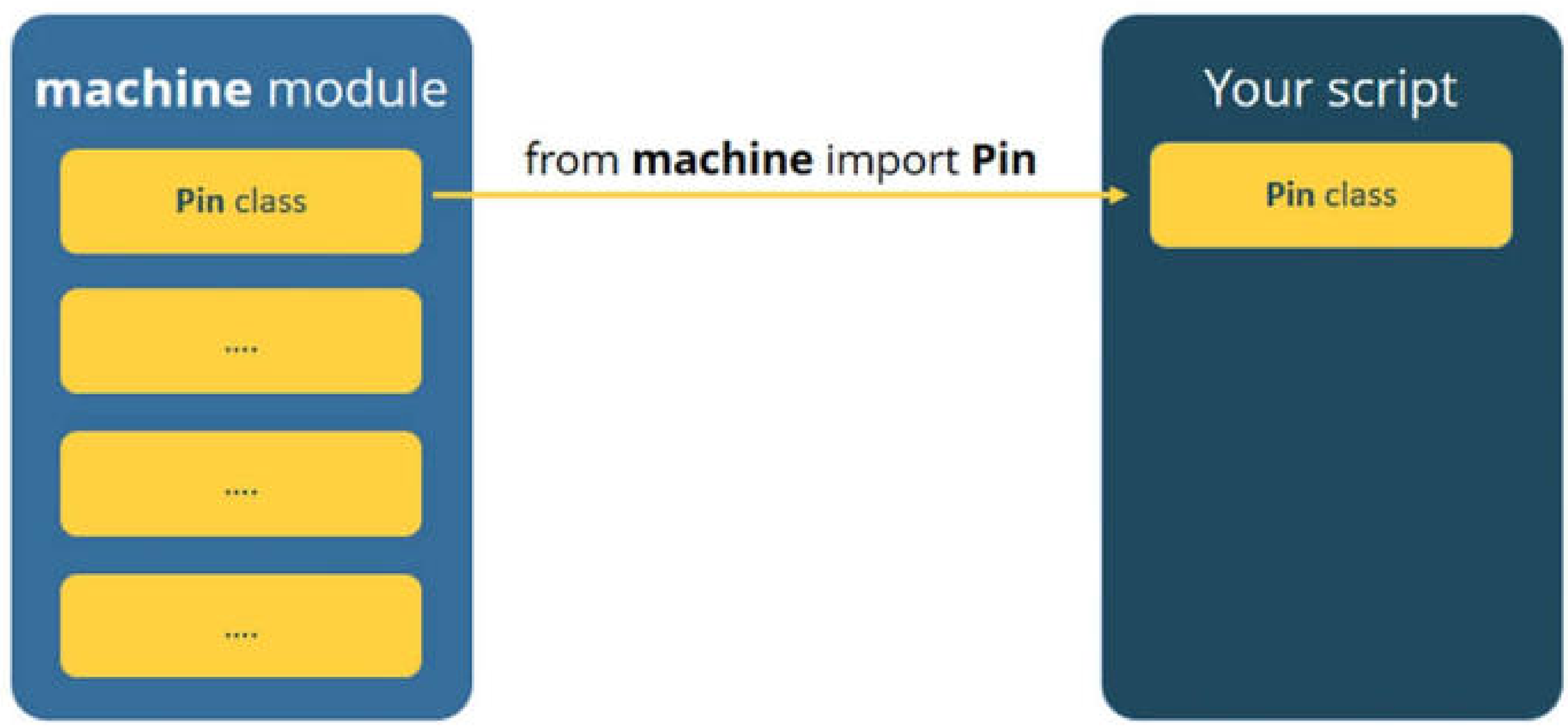
```
import module_name
```

For example, to import the `machine` library that contains classes to control GPIOs, type the following:

```
import machine
```

In most programs you won't need all the classes from one module. You may just want to import a single class. For example, to import only the `Pin` class from the `machine` module:

```
from machine import Pin
```



MicroPython vs C++ (Arduino IDE)

If you are used to program electronics using the Arduino IDE, you'll find that MicroPython has a much simpler and user-friendly syntax. Let's just summarize some of the main differences between your Arduino sketches and scripts in MicroPython:

- You don't use semicolon ; at the end of a statement
- After Boolean expressions in conditional statements and loops, you use a colon **:**
- To define code blocks use indentation, instead of curly brackets **{}**

- When creating a variable, you don't need to define which data type it is – you don't need to declare a variable
- Indentation in MicroPython is 2 spaces
- Python is an interpreted language and C++ is a compiled language

Conclusion

This was just a quick getting started guide to get you familiar with Python/MicroPython syntax. If you already know how to program in Python, programming in MicroPython is almost the same.

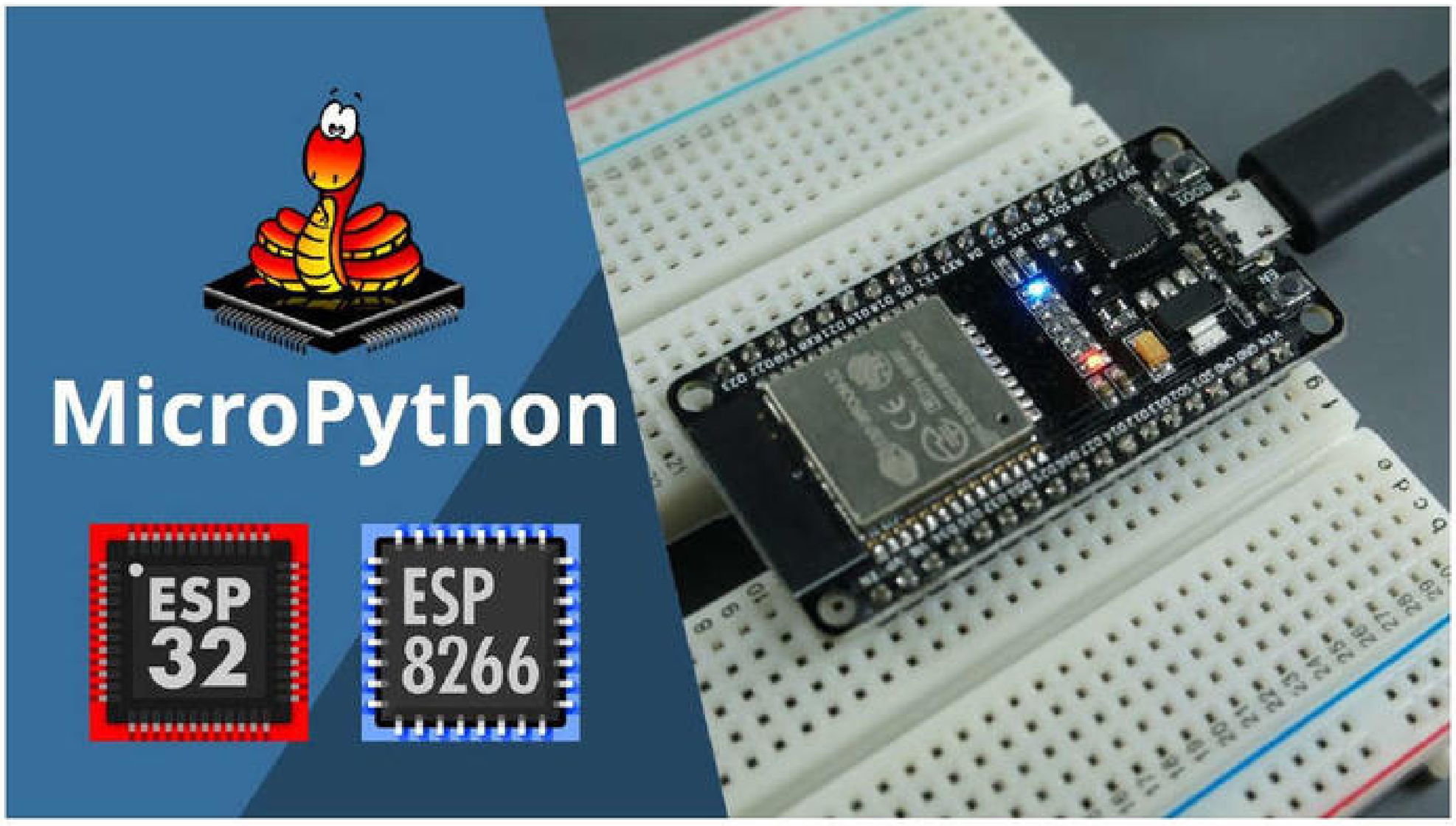
If you are used to program your electronics using Arduino IDE, you'll find that MicroPython is much simpler and less prone to syntax errors.

Finally, if this is your first-time programming digital electronics, we hope you enjoy MicroPython.

MODULE 3

Interacting with GPIOs

Blinking an LED



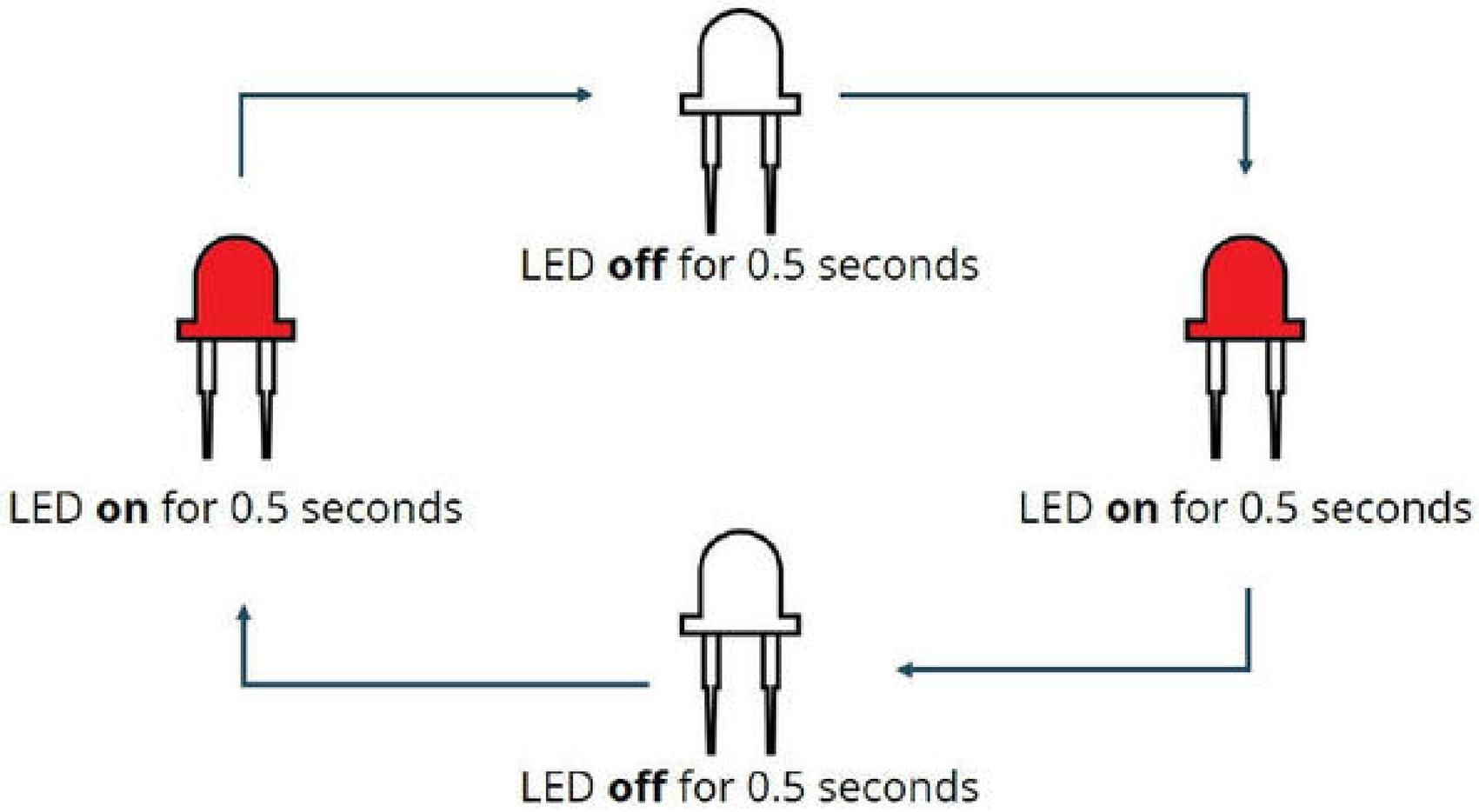
After all the theory we've been showing you, let's finally do something to interact with the physical world. The first thing we usually do when starting something new with electronics is building the blinking LED project. The blinking LED project is a great way to get users familiar with the programming language.

Project Overview

The project we'll build consists of simply blinking the ESP32/ESP8266 on-board LED. The on-board LED corresponds to GPIO 2 in both ESP32 and ESP8266. In simple terms, the blinking LED project works as follows:

- The LED turns on for 0.5 seconds—GPIO 2 set to HIGH.
- The LED turns off for 0.5 seconds—GPIO 2 set to LOW.
- The LED is on again for 0.5 seconds—GPIO 2 set to HIGH.
- The LED is off again for 0.5 seconds—GPIO 2 set to LOW.

This pattern continues until you tell the program to stop.



Script

The following script does precisely what we've described:

```
from machine import Pin
from time import sleep
led = Pin(2, Pin.OUT)
while True:
  led.value(not led.value())
  sleep(0.5)
```

SOURCE CODE
https://github.com/RuiSantosdotme/ESP-MicroPython/blob/master/code/Blink_LED/Blink_LED.py

How the Code Works

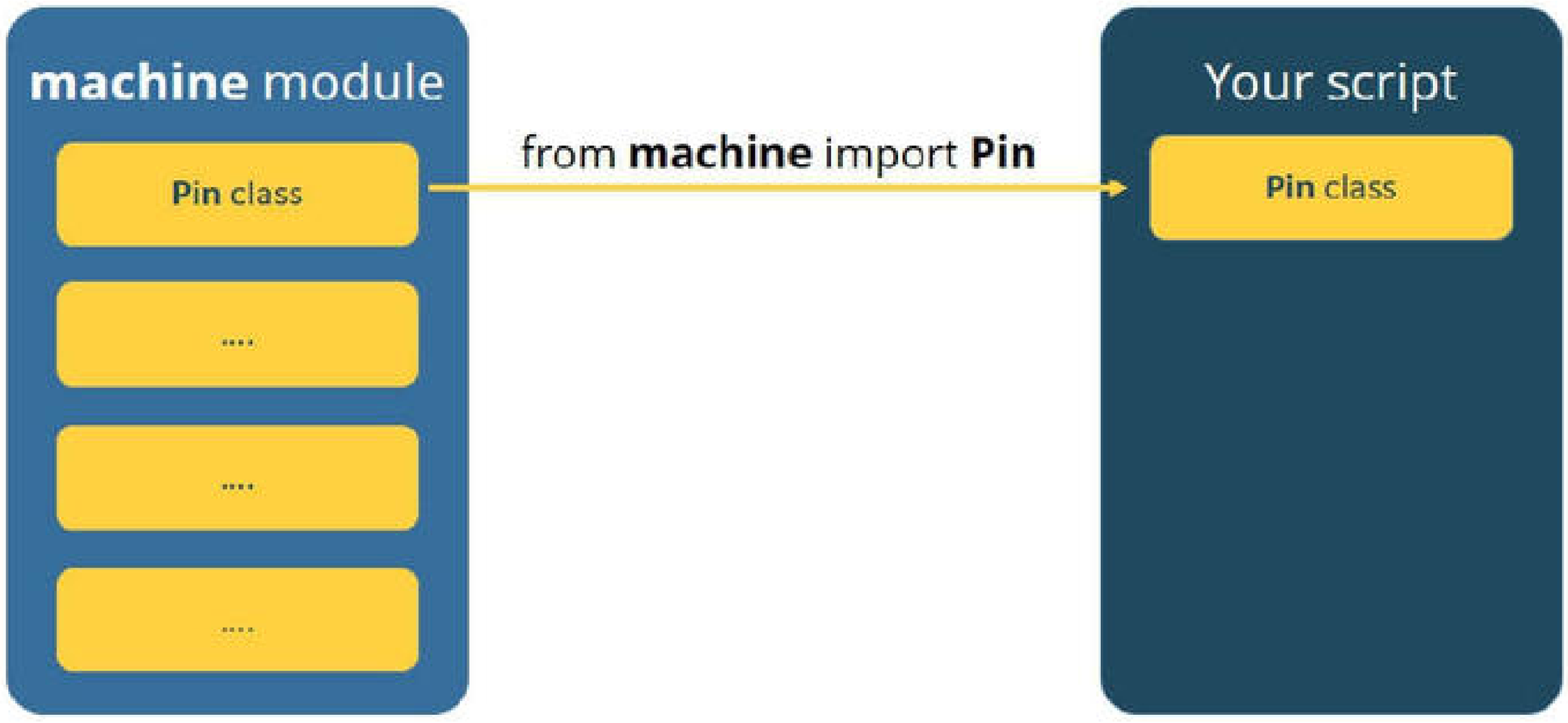
Let's take a closer look at the code to understand how it works.

The *machine* module

In all our scripts, we'll use the `machine` module. The `machine` module contains classes to interact with the physical world - this means classes to interact with the GPIOs.

The `machine` module contains many classes. We don't need all those classes in every script we create. In MicroPython, because we are writing code in constrained conditions, we should import only what we need. So, instead of importing all the

classes in the `machine` module, we just need to import the `Pin` class in this example.



The following line tells us to import only the `Pin` class from the `machine` module:

```
from machine import Pin
```

The *time* module

In most of our scripts, we'll use the `time` module. The `time` module, as the name suggests, allows us to deal with time:

- get time and date;
- use delays;
- measure how many seconds have passed since the program started;
- etc.

In this script, we need to add a delay to create the blinking effect. To add a delay we need to use the `sleep()` method. Because we only need that method from the `time` module, we import it as follows:

```
from time import sleep
```

Instantiating Pin objects

After importing all the necessary modules, we instantiate a `Pin` object called `led`.

The `Pin` object accepts these attributes in the following order:

```
(Pin number, pin mode, pull, value)
```

Here's what these arguments mean:

- **Pin number**: the GPIO number;
- **Pin mode**: a pin can be an input, output, or open-drain (we won't use open-drain mode in any of our examples):
    - **Pin.IN**
    - **Pin.OUT**
    - **Pin.OPEN_DRAIN**

Because we want GPIO 2 to act as an output, we need to set the mode to `Pin.OUT`.

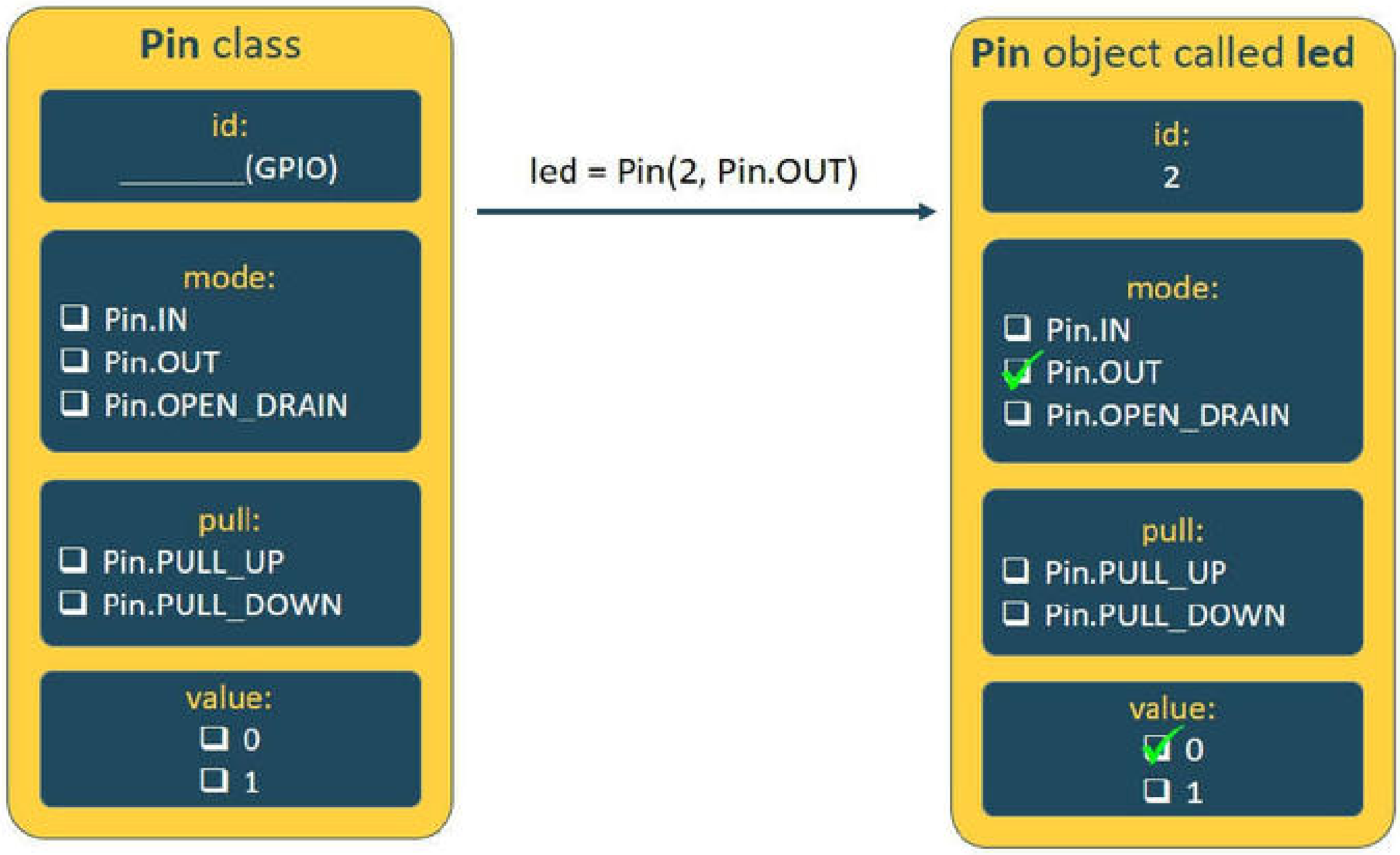
- **Pull**: this argument is used if we want to activate a pull up or pull-down internal resistor:
    - **Pin.PULL_UP** — pull-up
    - **Pin.PULL_DOWN** — pull-down

We won't set any internal resistor, so we don't need to pass this parameter.

- **Value**: can be 0 or 1, or True or False. Setting 0 or False means the GPIO is off. Setting 1 or True means the GPIO is on. If we don't pass any parameter, its state is 0 by default. So, our `led` object looks as follows:

```
led = Pin(2, Pin.OUT)
```

The following figure may help you better understand how to create our `led` object from the `Pin` class.



Then, you create a `while` loop that is always `True`. This is a little trick to make something run forever and ever.

```
while True:
```

> **Note:** if you are used to program with Arduino IDE, this `while` loop is similar to the `loop()` function in an Arduino sketch.

Inside the loop, we set if the GPIO is on or off. The `Pin` class provides a method called `value()` that allows us to change the state of the GPIO. For example, to change the value of the led, you use the following:

```
led.value(state)
```

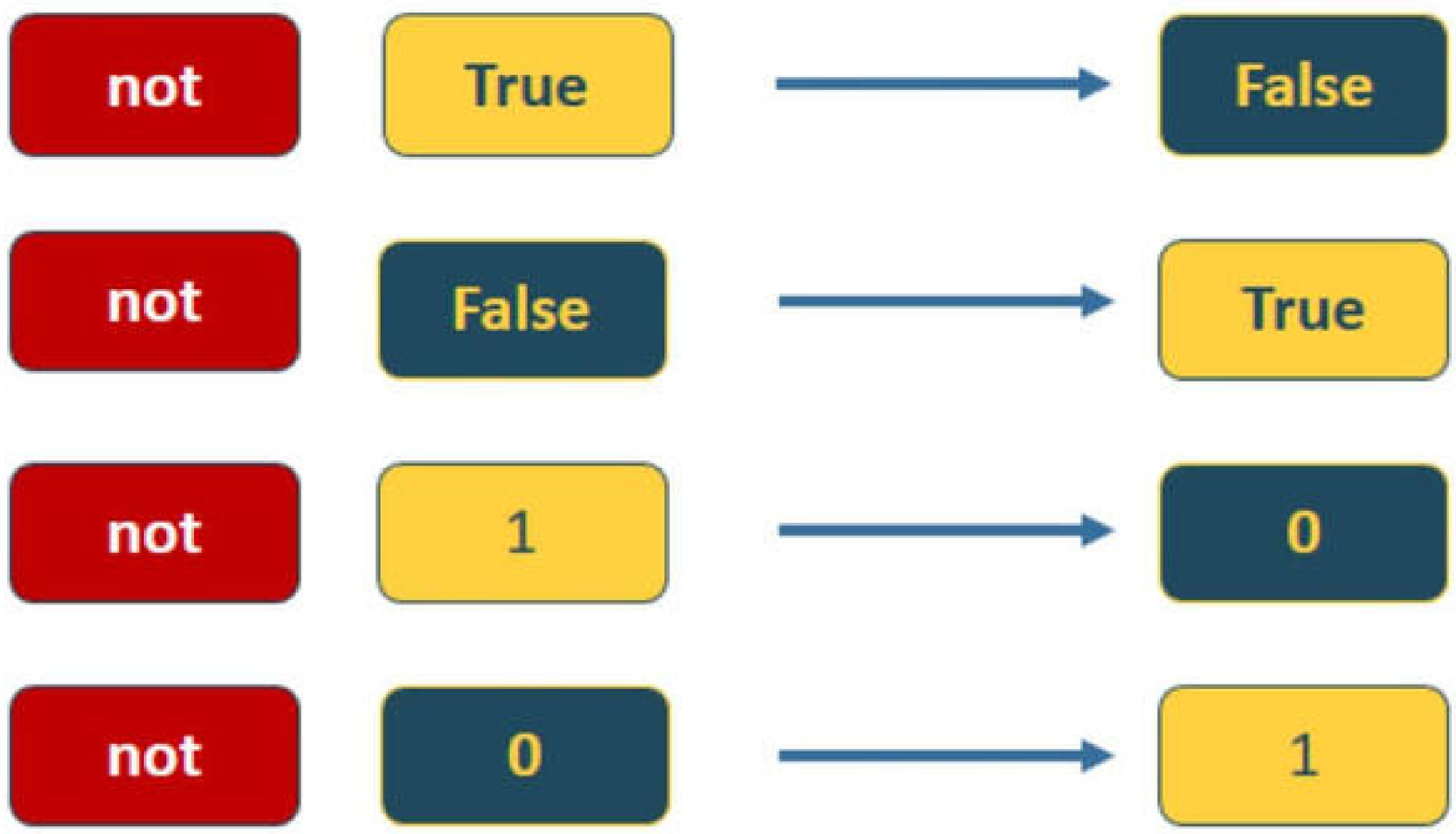
In which `state` corresponds to the state you want to set to the led: either 0 or 1 (or `True` or `False`).

As argument we pass the following expression:

```
not led.value()
```

The `led.value()` expression (without any arguments) returns the current state of the `led` object. In the first loop `led.value()` is 0. By applying the ***not*** logical operator, we get 1.

Not is a logical operator that reverses the state of the expression that follows. To better understand this concept, look at the following figure.



In the first loop, the `led.value(not led.value())` expression is the same as having `led.value(1)`. This sets the value of the `led` to 1 and turns GPIO 2 on. So, the on-board LED lights up.

After lighting up the LED, we wait for half a second (0.5 seconds).

```
sleep(0.5)
```

The `sleep()` function waits for the number of seconds you pass as argument. The code doesn't do anything else while you're in the `sleep()` function.

After waiting half a second, the loop restarts. This time, the `led.value()` is 1. By applying the `not` logical operator, it becomes 0. This is the same of writing `led.value(0)`, so the on-board LED turns off.

Then, we wait for half a second, and the loop repeats again and again, producing the blinking effect.

The following table may help you better understand what is going on in the code.

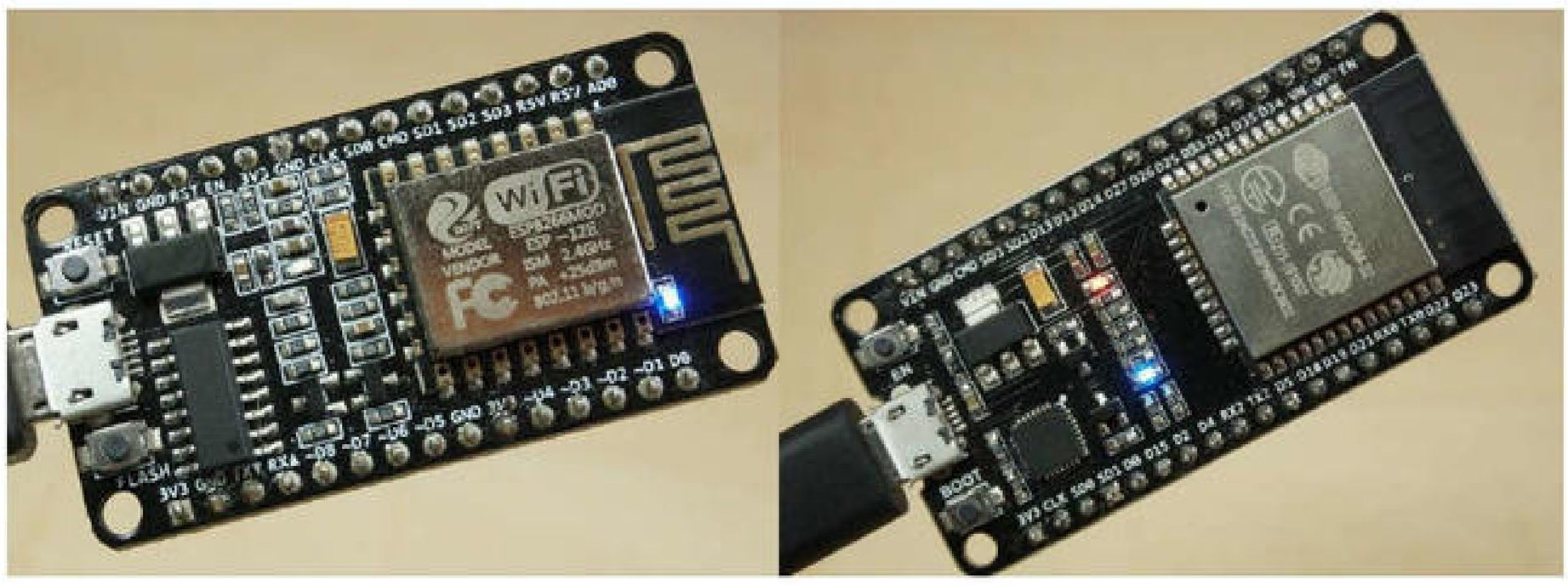
	led.value()	not led.value()	led.value(not led.value())
Loop number 1	0	1	led.value(1) → GPIO ON
Loop number 2	1	0	led.value(0) → GPIO OFF
Loop number 3	0	1	led.value(1) → GPIO ON
...	...	...	...

Upload the Code

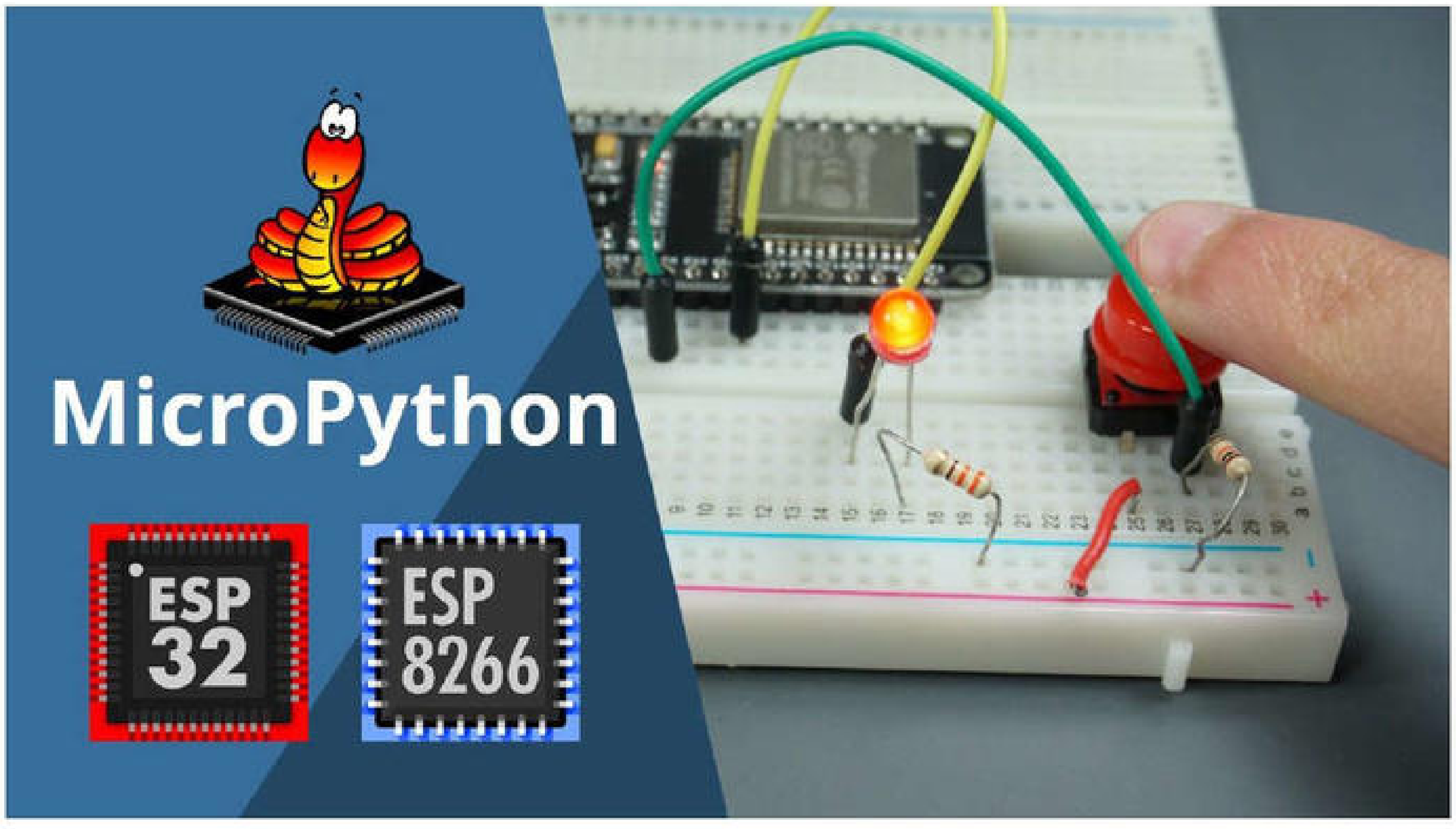
Open uPyCraft IDE. Connect your ESP board to your computer using an USB cable. Make sure you select the right COM port in **Tools ▸ Serial**, and the right board in **Tools ▸ Board**. Then, upload the code to your board as we've shown in you in previous Units.

Demonstration

The on-board ESP32 LED should be blinking every half a second.



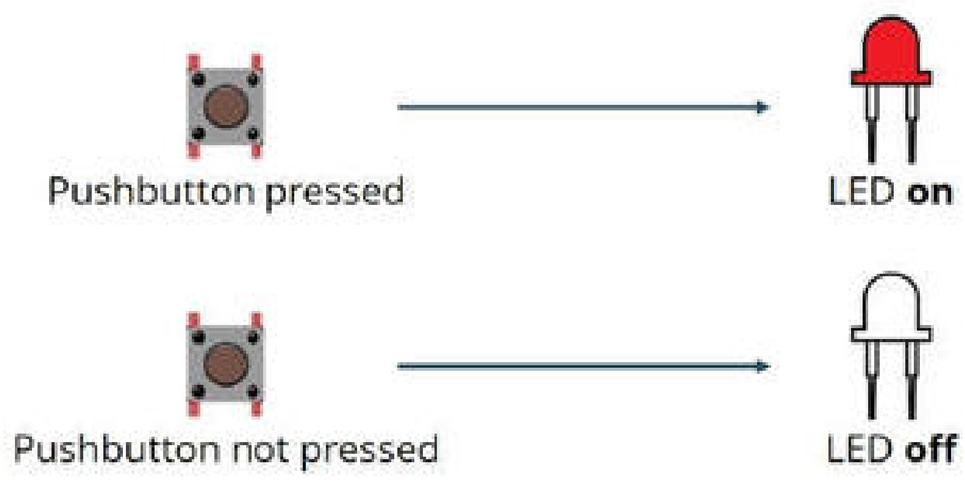
Digital Inputs and Digital Outputs



In the previous example, we've seen how to turn an LED on and off. Therefore, you already know how to control digital outputs. Now, we'll see how to read digital inputs. Reading digital inputs allows us to know the state of a GPIO. This is useful to know if a push button was pressed, for example.

Project Overview

In this Unit we'll read the pushbutton state and light up an LED accordingly.



To get the GPIO value, you need to use the `value()` method on a `Pin` object without passing any argument. For example, to get the state of a `Pin` object called `button`, use the following expression:

```
button.value()
```

Schematic

Before proceeding, you need to assemble a circuit with an LED and a pushbutton.

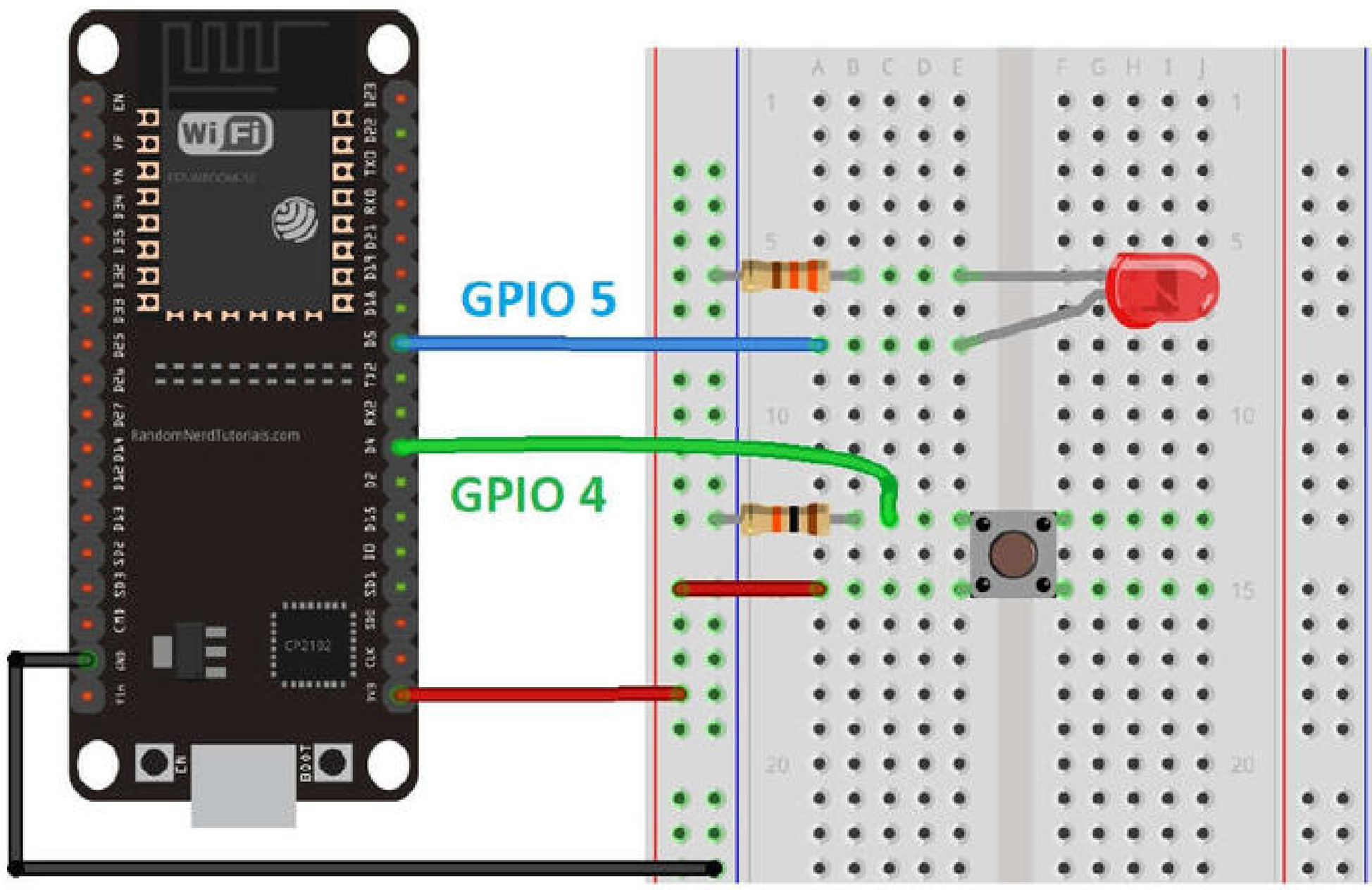
Parts required

Here's a list of parts you need to build the circuit:

- ESP32 or ESP8266
- 5mm LED
- 330 Ohm resistor
- Pushbutton
- 10k Ohm resistor
- Breadboard
- Jumper wires

Schematic – ESP32

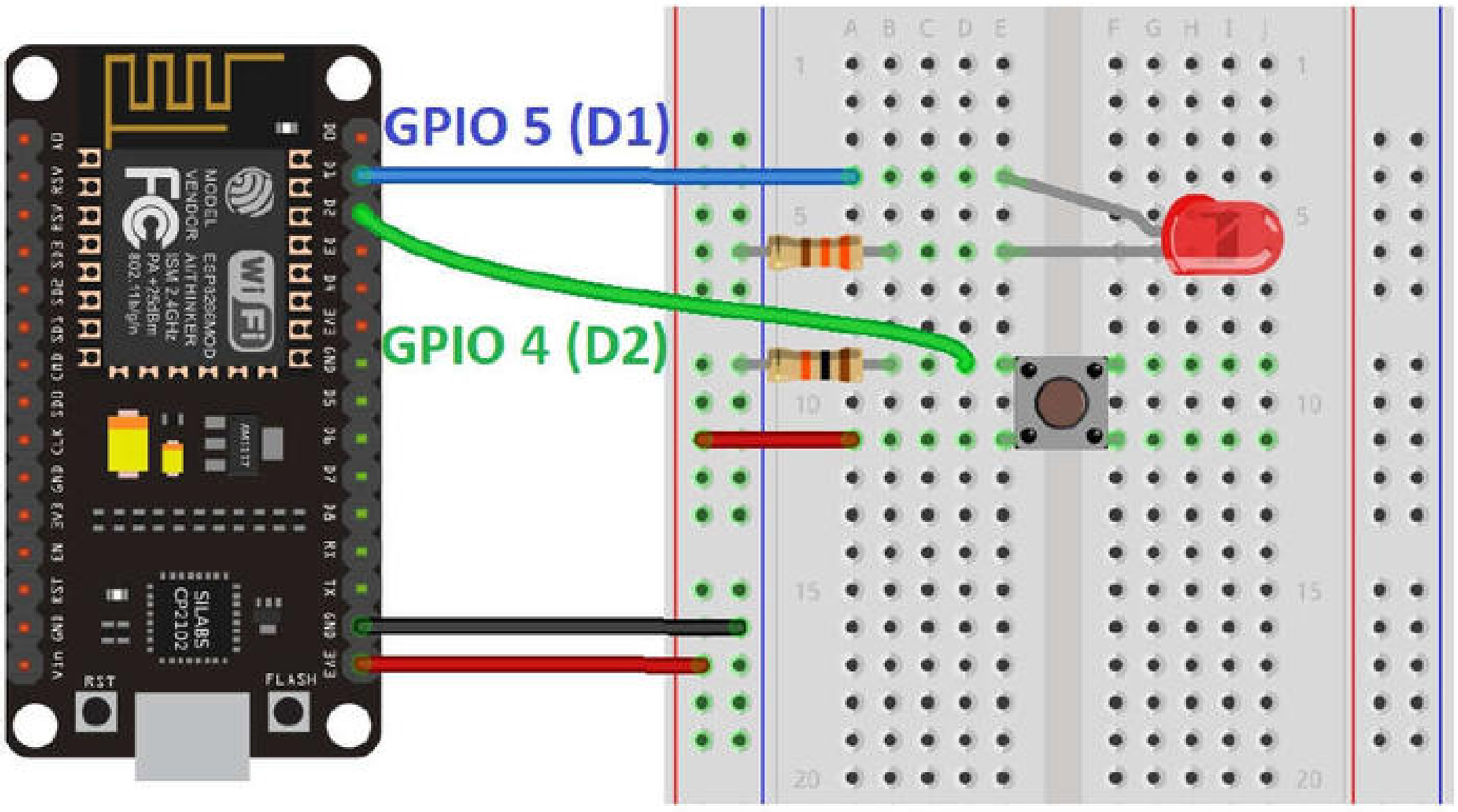
Follow the next schematic diagram if you're using an ESP32 board:



This schematic uses the ESP32 DEVKIT DOIT board version with 36 GPIOs. Before assembling the circuit double-check the pinout for the board you're using.

Schematic – ESP8266

Follow the next schematic diagram if you're using an ESP8266 board:



Script

The following code reads the state of the pushbutton and lights up the LED accordingly.

```
from machine import Pin
from time import sleep

led = Pin(5, Pin.OUT)
button = Pin(4, Pin.IN)

while True:
  led.value(button.value())
  sleep(0.1)
```

SOURCE CODE
https://github.com/RuiSantosdotme/ESP-MicroPython/blob/master/code/GPIOs/Pushbutton/pushbutton_led.py

How the Code Works

As explained previously, import the `Pin` class from the `machine` module, and the `sleep` class from the `time` module.

```
from machine import Pin
from time import sleep
```

Then, create a `Pin` object called `led` on GPIO 5. LEDs are outputs, so pass as second argument `Pin.OUT`. We also create a button on GPIO 4. Buttons are inputs, so use `Pin.IN`.

```
led = Pin(5, Pin.OUT)
button = Pin(4, Pin.IN)
```

Use `button.value()` to return/read the button state. We pass the `button.value()` expression as an argument to the LED value.

```
led.value(button.value())
```

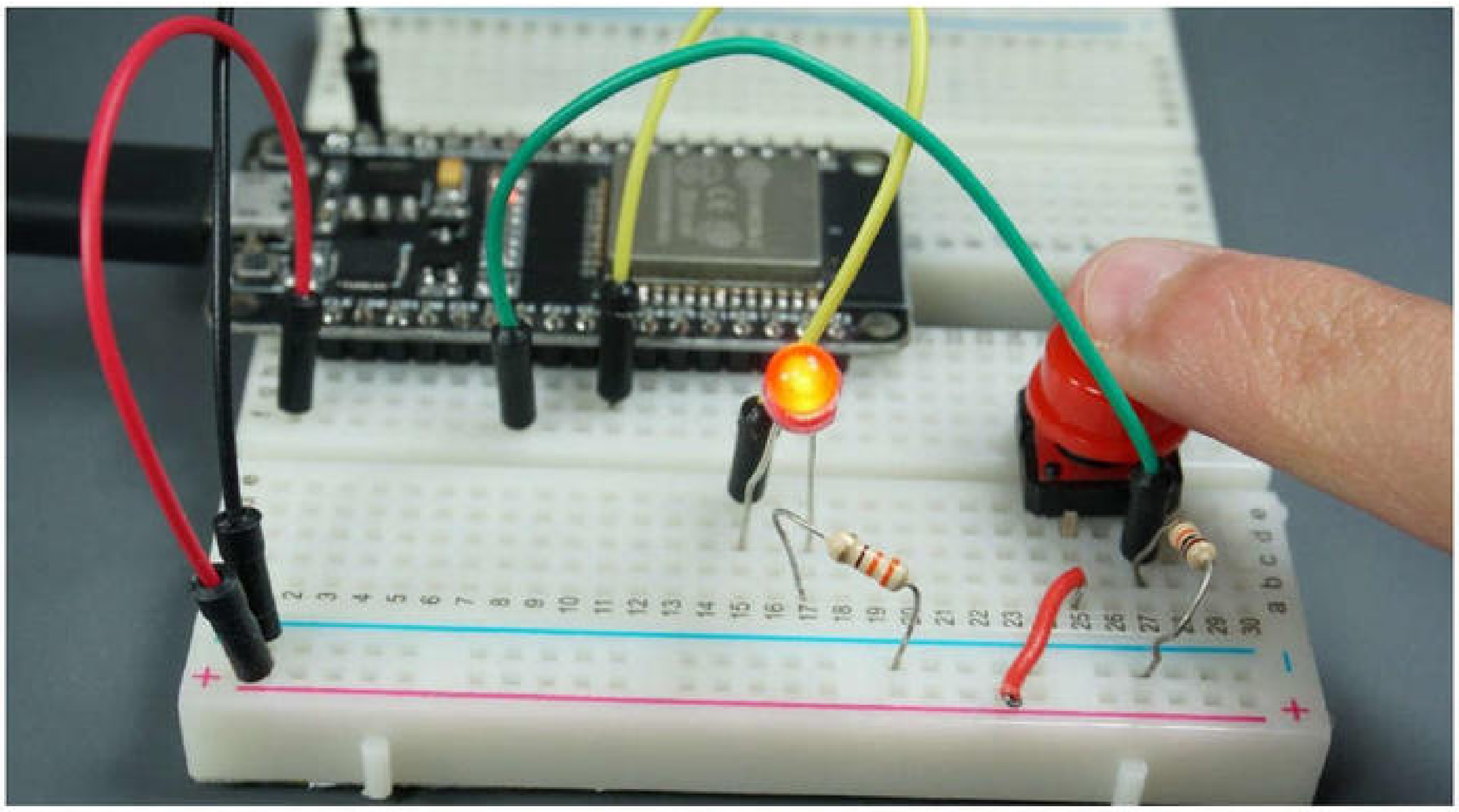
This way, when we press the button, `button.value()` returns 1. So, this is the same as having `led.value(1)`. This sets the LED state to 1, lighting up the LED. When the pushbutton is not being pressed, `button.value()` returns 0. So, we have `led.value(0)`, and the LED stays off.

The following table may help you understand what's happening in the code.

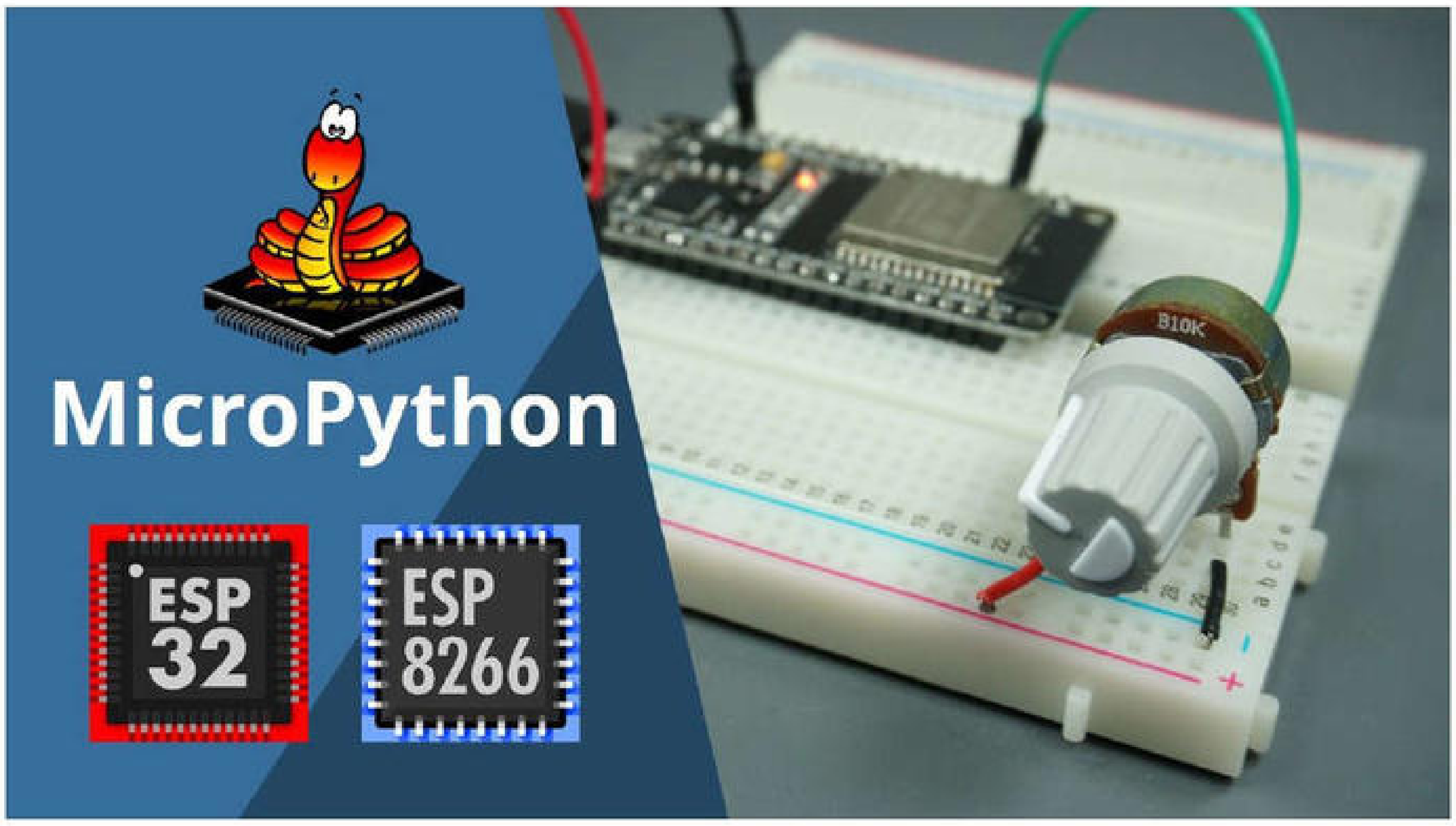
	button.value()	**led.value(button.value())**	**GPIO state**
button not pressed	0	led.value(0)	GPIO OFF
button pressed	1	led.value(1)	GPIO ON

Demonstration

Upload the code to your board. Then, the LED should light up when you press the button and turn off when you release it.



Analog Inputs



In this Unit, you'll learn how to read an analog input. This is useful to read values from variable resistors like potentiometers, or analog sensors.

Project Overview

In the previous Unit you've learned how to detect if a GPIO is on (3.3V) or off (0V). Using ADC (analog to digital converter) GPIOs allow you to read voltages between 0V and 3.3V.

The voltage measured is then assigned to a value between 0 and 4095 (in case of the ESP32), or between 0 and 1023 (in case of the ESP8266) in which 0V corresponds to 0, and 3.3V corresponds to 4095 (or 1023 for the ESP8266). Any voltage between 0V and 3.3V is given the corresponding value in between.

In this section, we'll build a simple example that reads the analog values from a potentiometer.

Analog reading with ESP8266

ESP8266 only has one analog pin called ADC 0. The ESP8266 analog pin has 10-bit resolution. It reads the voltage from 0 to 3.3V and then, assigns a value between 0 and 1023.

Note: some versions of the ESP8266 only read a maximum of 1V on the ADC pin. Make sure you don't exceed the maximum recommended voltage for your board.

Analog reading with ESP32

There are several pins on the ESP32 that can act as analog pins - these are called ADC pins. All the following GPIOs can act as ADC pins: 0, 2, 4, 12, 13, 14, 15, 25, 26, 27, 32, 33, 34, 35, 36, and 39.

ESP32 ADC pins have 12-bit resolution by default. These pins read voltage between 0 and 3.3V and then return a value between 0 and 4095. The resolution can be changed on the code. For example, you may want to have just 10-bit resolution to get a value between 0 and 1023.

The following table shows some differences between analog reading on the ESP8266 and the ESP32 board.

	ESP8266	**ESP32**
Analog pins	ADC 0	GPIOs: 0, 2, 4, 12, 13, 14, 15, 25, 26, 27, 32, 33, 34, 35, 36, and 39.
Resolution	10-bit (0-1023)	12-bit (0-4095)
Change resolution in the code	No	Yes

Analog reading works differently in ESP32 and ESP8266. There is a different schematic and a different script for each board.

Schematic

Before proceeding with this Unit, you need to connect a potentiometer to your ESP32 or ESP8266 board.

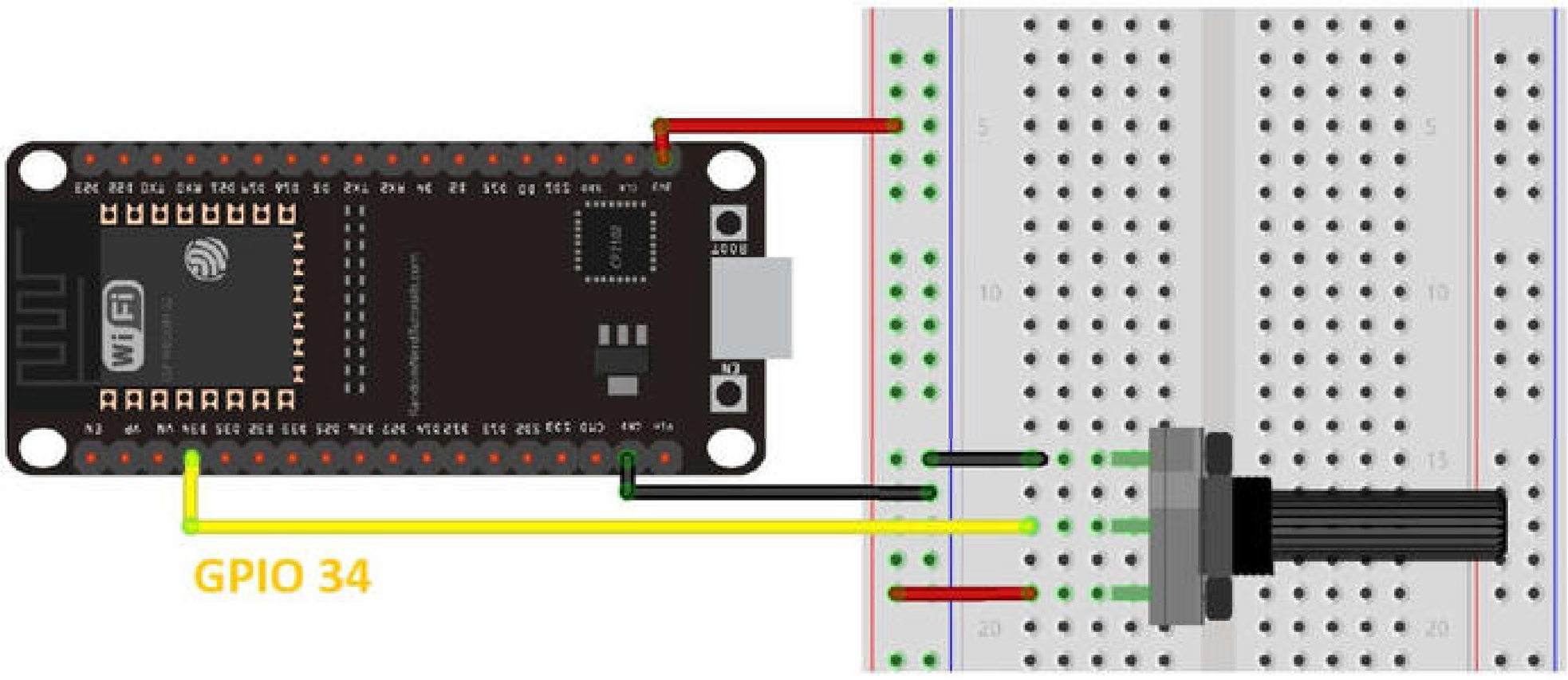
Parts required

Here's a list of parts you need to build the circuit:

- ESP32 or ESP8266
- Potentiometer
- Breadboard
- Jumper wires

Schematic – ESP32

Follow the next schematic diagram if you're using an ESP32 board:

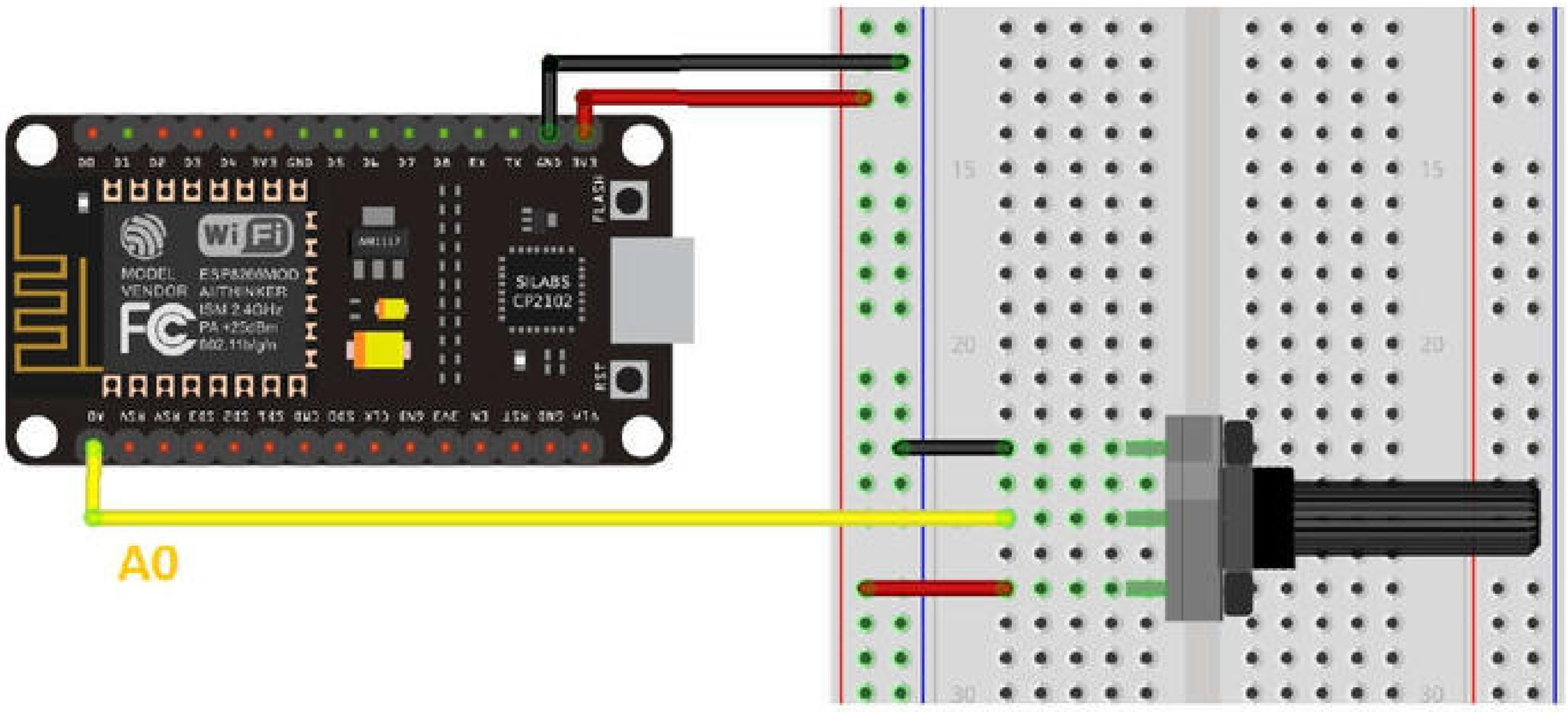


This schematic uses the ESP32 DEVKIT DOIT board version with 36 GPIOs. Before assembling the circuit double-check the pinout for the board you're using.

Note: in this example we're using GPIO 34 to read analog values from the potentiometer, but you can choose any other GPIO that supports ADC.

Schematic – ESP8266

Follow the next schematic diagram if you're using an ESP8266 board:



Script

There are a few differences when it comes to analog reading in ESP32 and ESP8266 regarding the code. You should write a slightly different script depending on the board you're using. Make sure you follow the code for your specific board.

Script - ESP32

The following script for the ESP32 reads analog values from GPIO 34.

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

pot = ADC(Pin(34))
pot.atten(ADC.ATTN_11DB)  #Full range: 3.3v

while True:
  pot_value = pot.read()
  print(pot_value)
  sleep(0.1)
```

SOURCE CODE
https://github.com/RuiSantosdotme/ESP-MicroPython/blob/master/code/GPIOs/Analog_Read_Pot/read_pot_esp32.py

How the code works

To read analog inputs, import the `ADC` class in addition to the `Pin` class from the `machine` module. We also import the `sleep` method.

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

Then, create an `ADC` object called `pot` on GPIO 34.

```
pot = ADC(Pin(34))
```

The following line defines that we want to be able to read voltage in full range.

```
pot.atten(ADC.ATTN_11DB)
```

This means we want to read voltage from 0 to 3.3V. This corresponds to setting the attenuation ratio of 11db. For that, we use the `atten()` method and pass as argument: `ADC.ATTN_11DB`

The `atten()` method can take the following arguments:

- **ADC.ATTN_0DB** — the full range voltage: 1.2V
- **ADC.ATTN_2_5DB** — the full range voltage: 1.5V
- **ADC.ATTN_6DB** — the full range voltage: 2.0V
- **ADC.ATTN_11DB** — the full range voltage: 3.3V

In the `while` loop, read the `pot` value and save it in the `pot_value` variable. To read the value from the pot, simply use the `read()` method on the `pot` object.

```
pot_value = pot.read()
```

Then, print the pot value.

```
print(pot_value)
```

At the end, add a delay of 100 ms.

```
sleep(0.1)
```

When you rotate the potentiometer, you get values from 0 to 4095 - that's because the ADC pins have a 12-bit resolution by default. You may want to get values in

other ranges. You can change the resolution using the `width()` method as follows:

```
ADC.width(bit)
```

The `bit` argument can be one of the following parameters:

- **ADC.WIDTH_9BIT**: range 0 to 511
- **ADC.WIDTH_10BIT**: range 0 to 1023
- **ADC.WIDTH_11BIT**: range 0 to 2047
- **ADC.WIDTH_12BIT**: range 0 to 4095

For example:

```
ADC.width(ADC.WIDTH_12BIT)
```

In summary:

- To read an analog value you need to import the `ADC` class;
- To create an ADC object simply use `ADC(Pin(GPIO))`, in which GPIO is the number of the GPIO you want to read the analog values;
- To read the analog value, simply use the `read()` method on the `ADC` object.

Script – ESP8266

The following script for the ESP8266 reads analog values from ADC 0 pin.

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

pot = ADC(0)

while True:
  pot_value = pot.read()
  print(pot_value)
  sleep(0.1)
```

SOURCE CODE
https://github.com/RuiSantosdotme/ESP-MicroPython/blob/master/code/GPIOs/Analog_Read_Pot/read_pot_esp8266.py

How the code works

To read analog inputs, import the `ADC` class in addition to the `Pin` class from the `machine` module. We also import the `sleep` method.

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

Then, create an `ADC` object called `pot` on ADC 0 pin.

```
pot = ADC(0)
```

Note: ADC0 (A0) is the only pin on the ESP8266 that supports analog reading.

In the loop, read the pot value and save it in the `pot_value` variable. To read the value from the `pot`, use the `read()` method on the `pot` object.

```
pot_value = pot.read()
```

Then, print the `pot_value`.

```
print(pot_value)
```

At the end, add a delay of 100 ms.

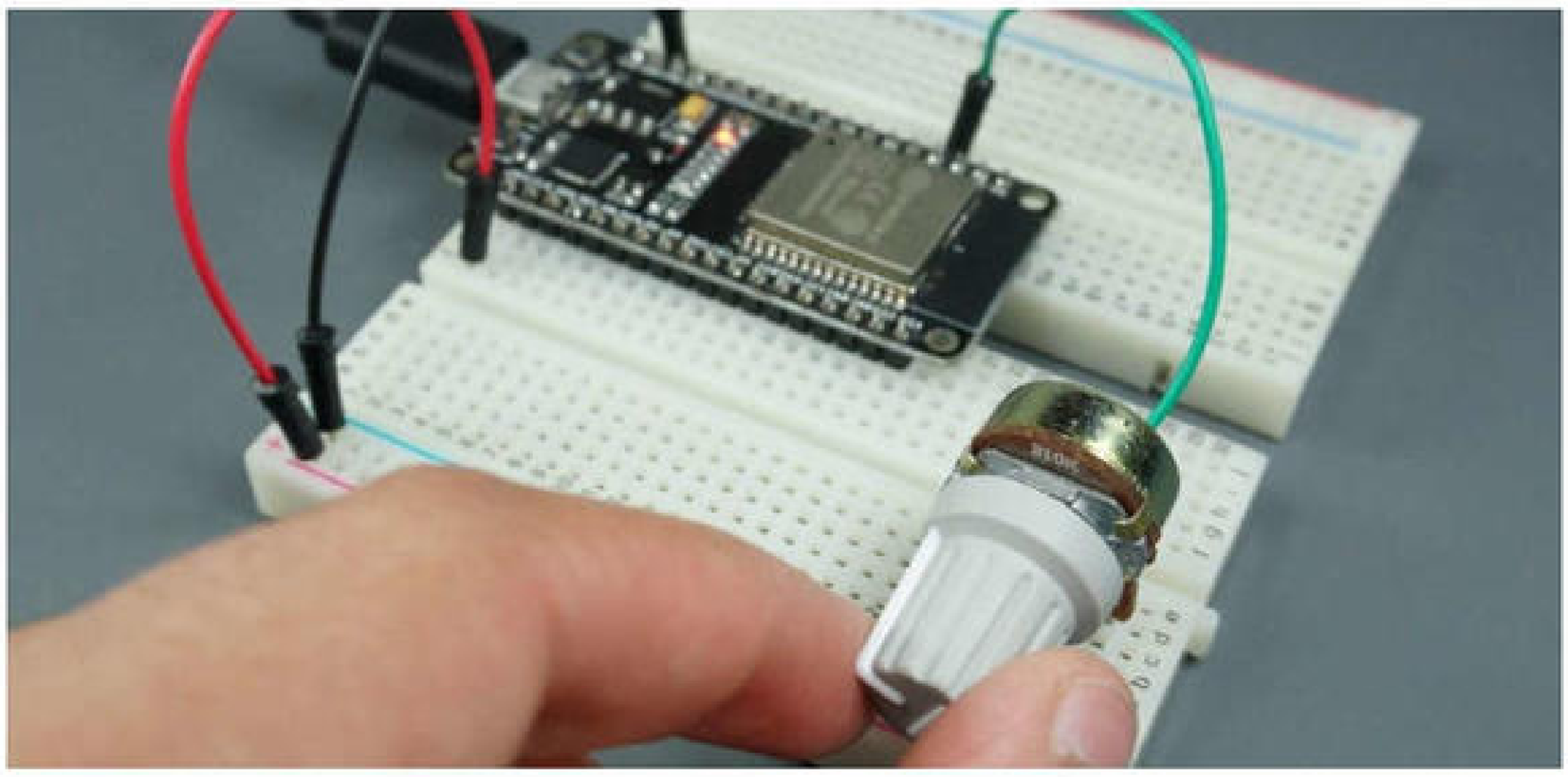
```
sleep(0.1)
```

In summary:

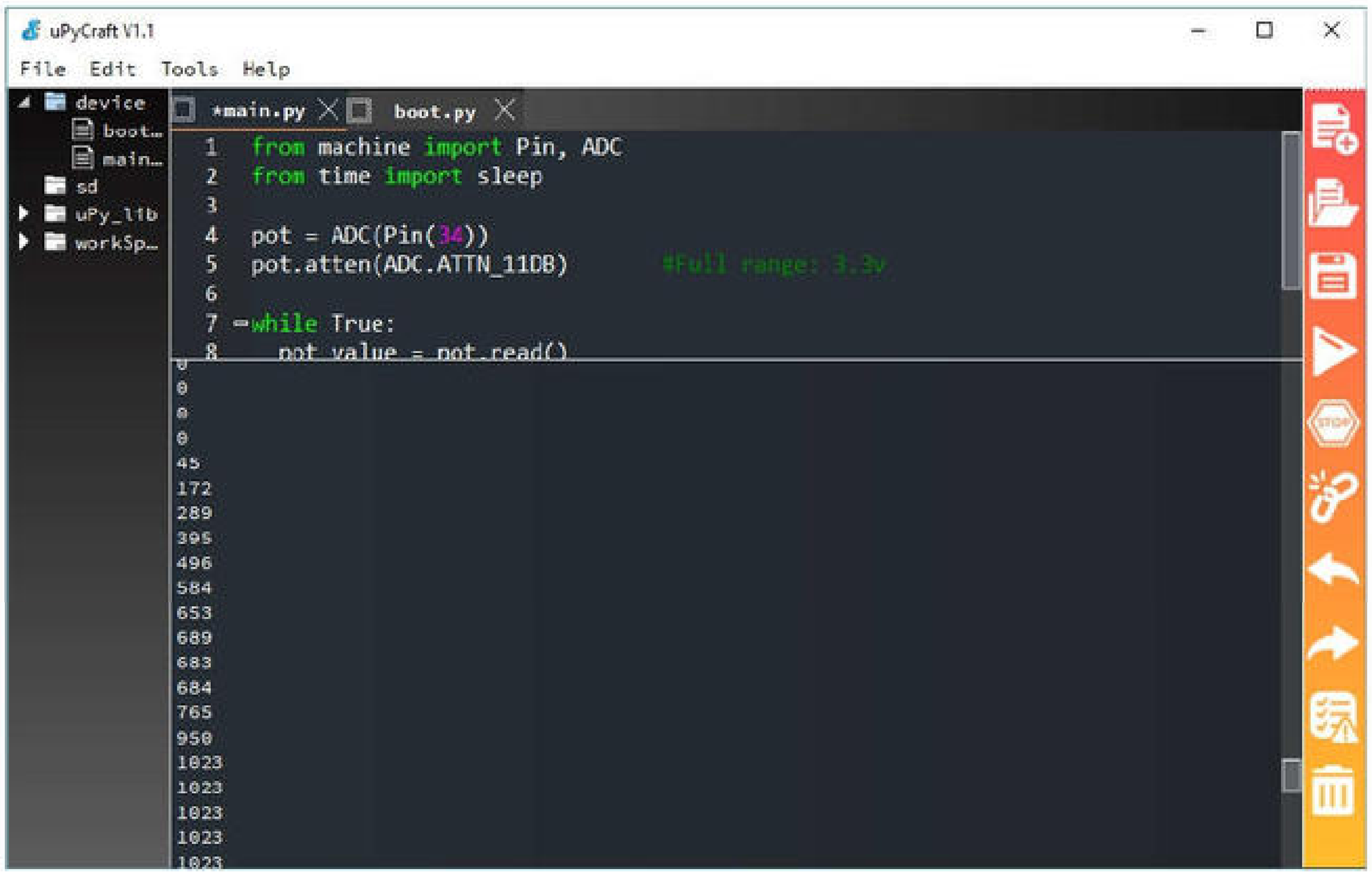
- To read an analog value you use the `ADC` class;
- To create an ADC object simply call `ADC(0)`. The ESP8266 only supports ADC reading on ADC 0 pin.
- To read the analog value, use the `read()` method on the `ADC` object.

Demonstration

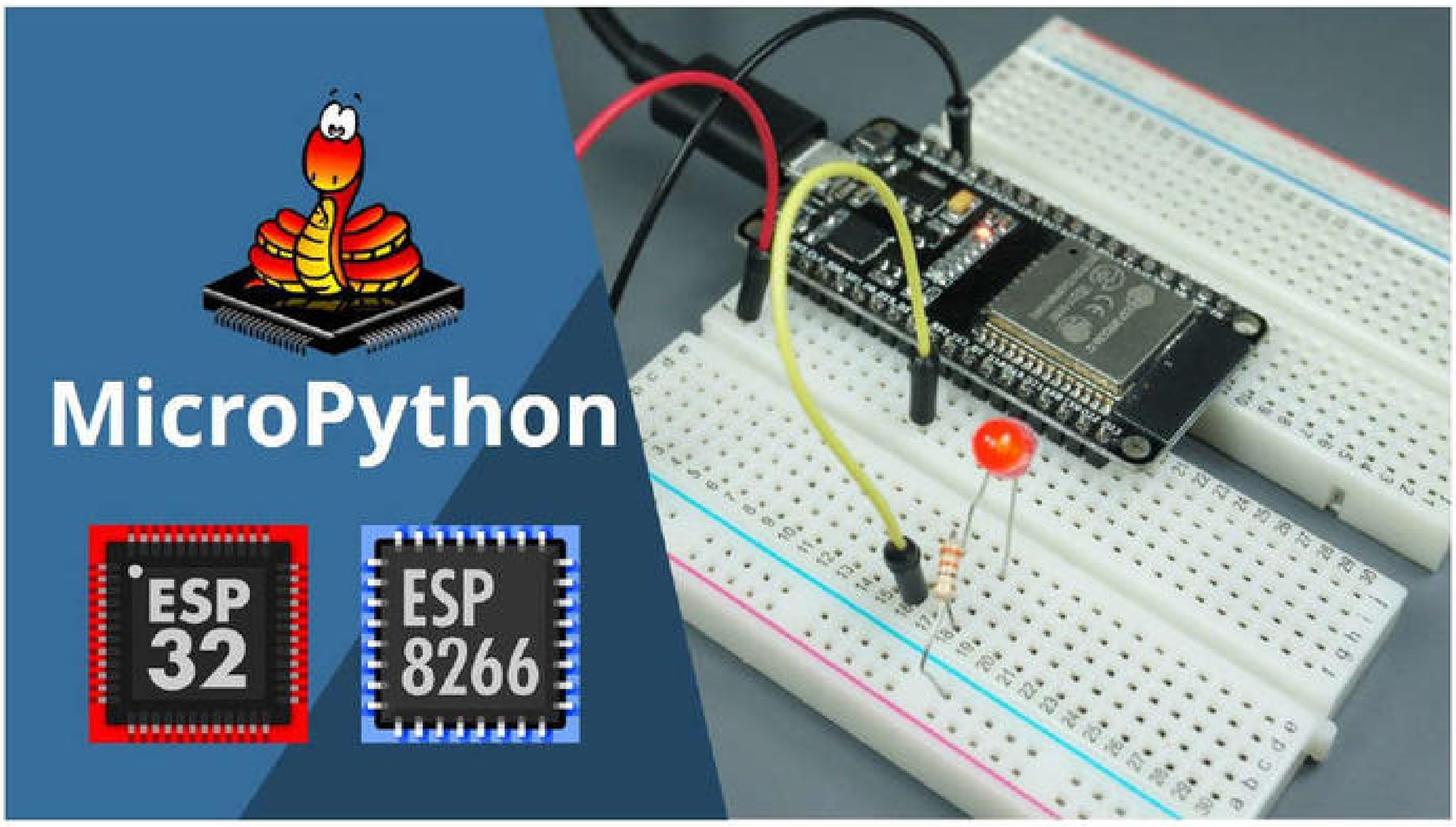
After uploading the code to your board, rotate the potentiometer.



In the uPyCraft Shell, if you're using an ESP32 you should get readings between 0 and 4095 — or readings between 0 and 1023 with an ESP8266.



PWM – Pulse Width Modulation

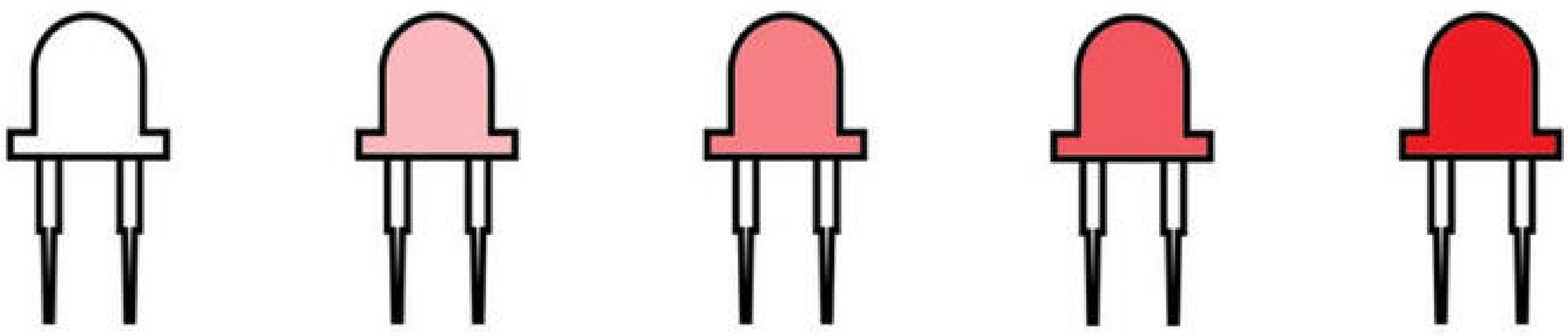


In this Unit, you'll learn how to dim an LED using PWM. We'll also build a simple project in which you can control the LED brightness using a potentiometer.

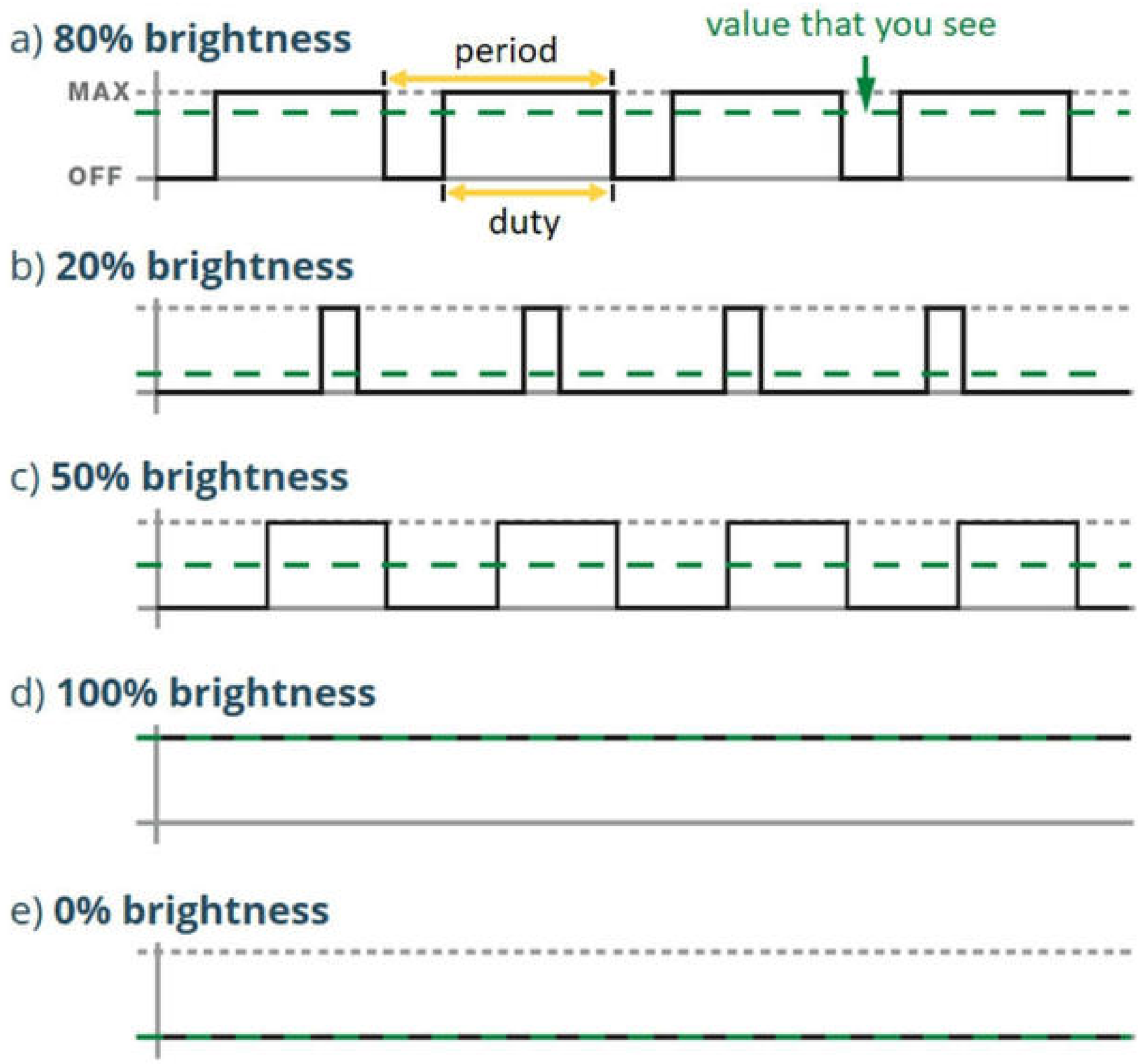
Pulse-Width Modulation

The ESP32/ESP8266 GPIOs can be set either to output 0V or 3.3V, but they can't output any voltages in between (apart from the DAC pins on the ESP32). However, you can output "fake" mid-level voltages using pulse-width modulation (PWM), which is how you'll produce varying levels of LED brightness for this project.

If you alternate an LED's voltage between HIGH and LOW very fast, your eyes can't keep up with the speed at which the LED switches on and off; you'll simply see some gradations in brightness.



That's basically how PWM works—by producing an output that changes between HIGH and LOW at a very high frequency. The duty cycle is the fraction of the time period at which the LED is set to HIGH. The following figure illustrates how PWM works.



For instance, a duty cycle of 50 percent results in 50 percent LED brightness, a duty cycle of 0 means the LED is fully off, and a duty cycle of 100 means the LED is fully on. Changing the duty cycle is how you produce different levels of brightness, and that's what we're going to do here.

Schematic

For this example, wire an LED to your ESP board. We'll wire the LED to GPIO 5 in both boards, but you can choose another suitable PWM pin. All ESP pins that can act as outputs can generate PWM signals.

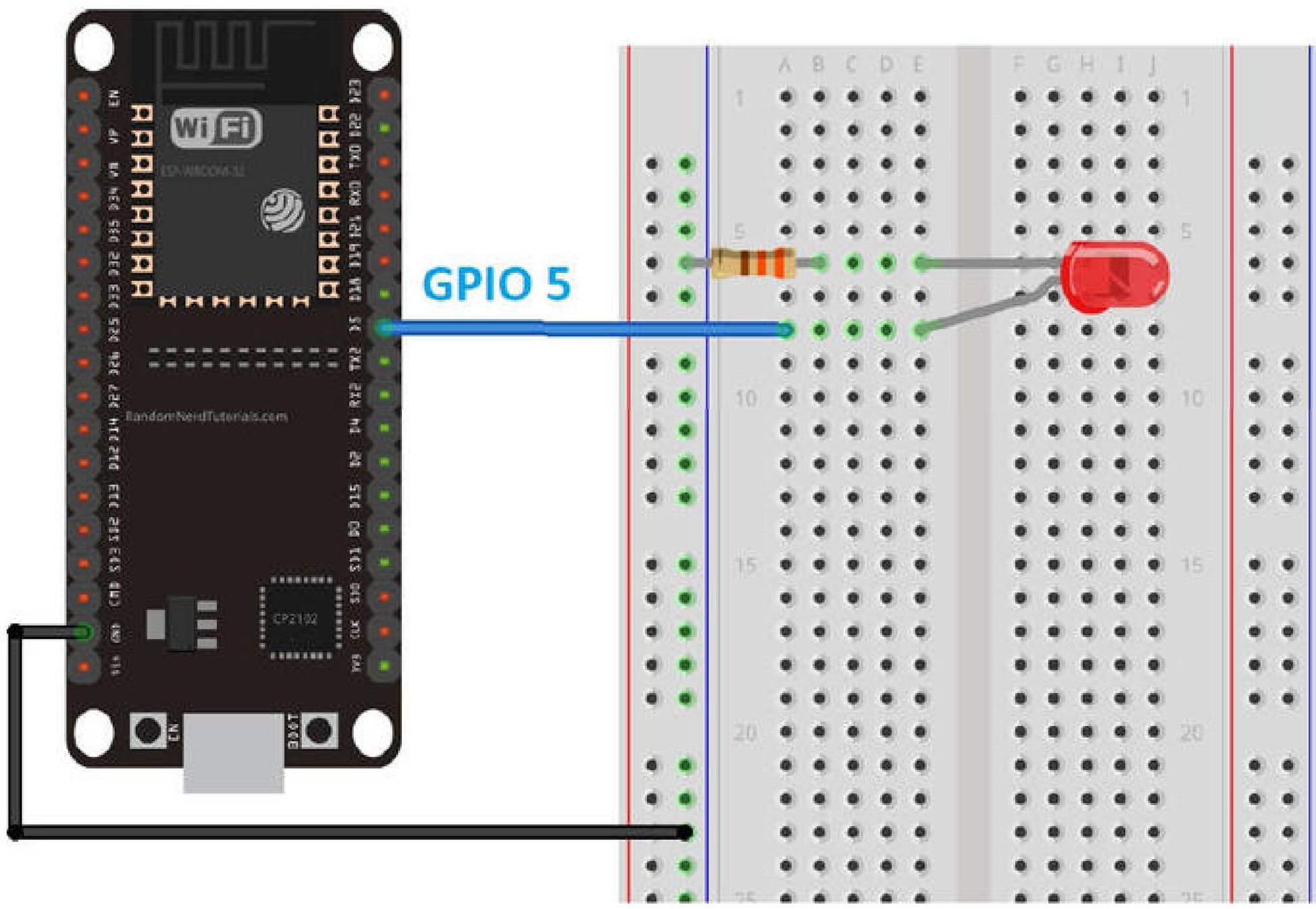
Parts required

Here's a list of parts you need to build the circuit:

- ESP32 or ESP8266
- 5mm LED
- 330 Ohm resistor
- Breadboard
- Jumper wires

Schematic – ESP32

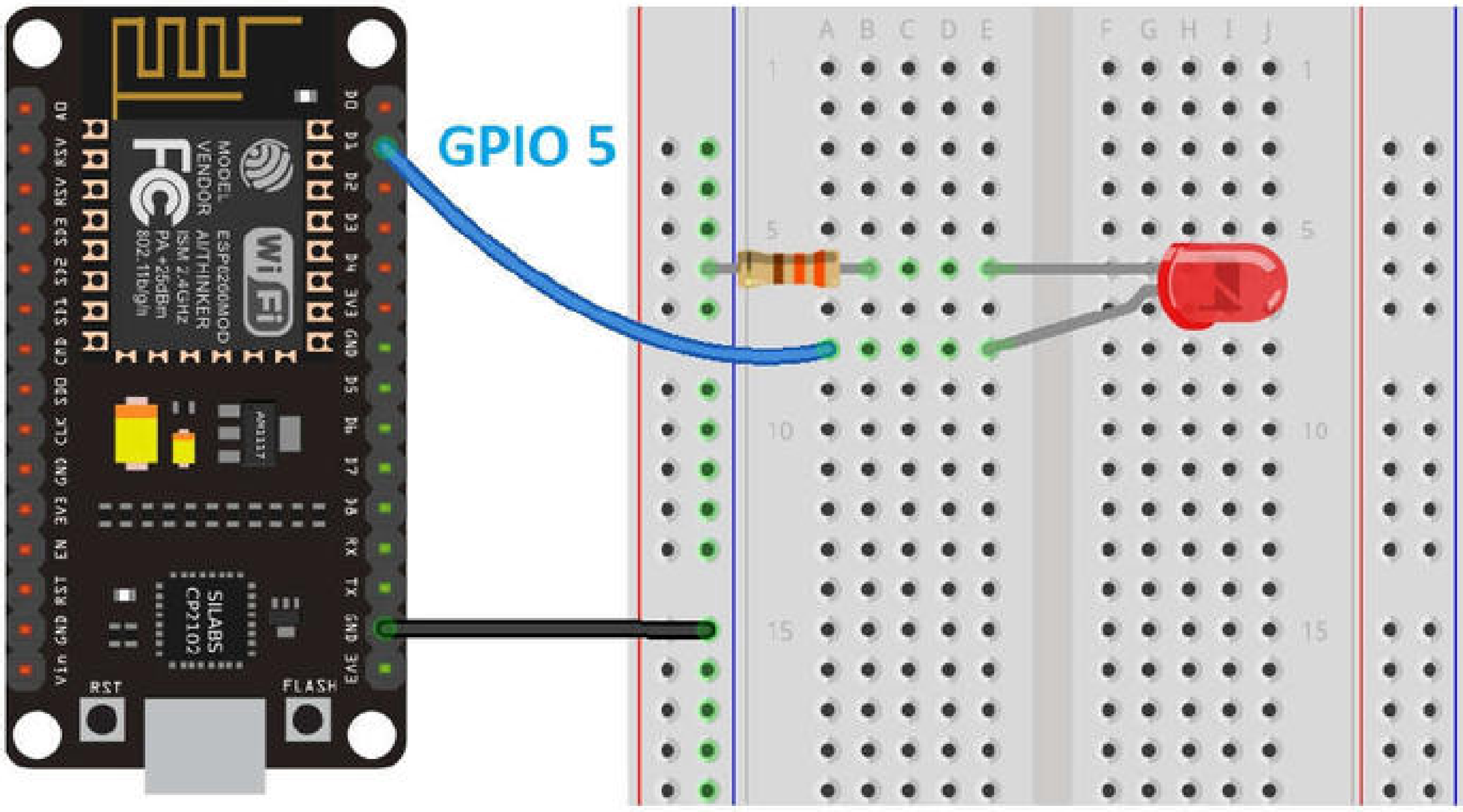
Follow the next schematic diagram if you're using an ESP32 board:



(This schematic uses the ESP32 DEVKIT DOIT board version with 36 GPIOs. Before assembling the circuit double-check the pinout for the board you're using.)

Schematic – ESP8266

Follow the next schematic diagram if you're using an ESP8266 board:



Script

Here's the script that changes the LED brightness over time by increasing the duty cycle. This script works with the ESP32 and ESP8266.

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

frequency = 5000
led = PWM(Pin(5), frequency)

while True:
  for duty_cycle in range(0, 1024):
    led.duty(duty_cycle)
    sleep(0.005)
```

SOURCE CODE

https://github.com/RuiSantosdotme/ESP-MicroPython/blob/master/code/GPIOs/PWM/pwm_basic.py

How the code works

To create a PWM pin, import the `PWM` class in addition to the `Pin` class from the `machine` module.

```
from machine import Pin, PWM
```

Then, create a `PWM` object called `led`.

```
led = PWM(Pin(5), frequency)
```

To create a `PWM` object, you need to pass as parameters, the pin it is connected to, the signal frequency and the duty cycle.

- **Frequency**: The frequency can be a value between 0 and 78125. A frequency of 5000 Hz can be used to control an LED brightness.
- **Duty cycle**: The duty cycle can be a value between 0 and 1023. In which 1023 corresponds to 100% duty cycle (full brightness), and 0 corresponds to 0% duty cycle (unlit LED).

We'll just set the duty cycle on the `while` loop, so we don't need to pass the duty cycle parameter. If you don't set the duty cycle when instantiating the `PWM` object, it will be 0 by default.

To set the duty cycle use the `duty()` method on the `PWM` object and pass the duty cycle as an argument:

```
led.duty(duty_cycle)
```

Inside the `while` loop, we create a `for` loop that increases the duty cycle by 1 in each loop with an interval of 5 ms between each change.

The `range()` function has the following syntax:

```
range(start, stop)
```

- **Start**: a number that specifies at which position to start. We want to start with 0 duty cycle;
- **Stop**: a number that specifies at which position we want to stop, excluding that value. The maximum duty cycle is 1023, because we are incrementing 1 in each loop, the last value should be 1023+1. So, we'll use 1024.

In each `for` loop, we print the current value of the duty cycle and set the LED's duty cycle to the current `duty_cycle` value:

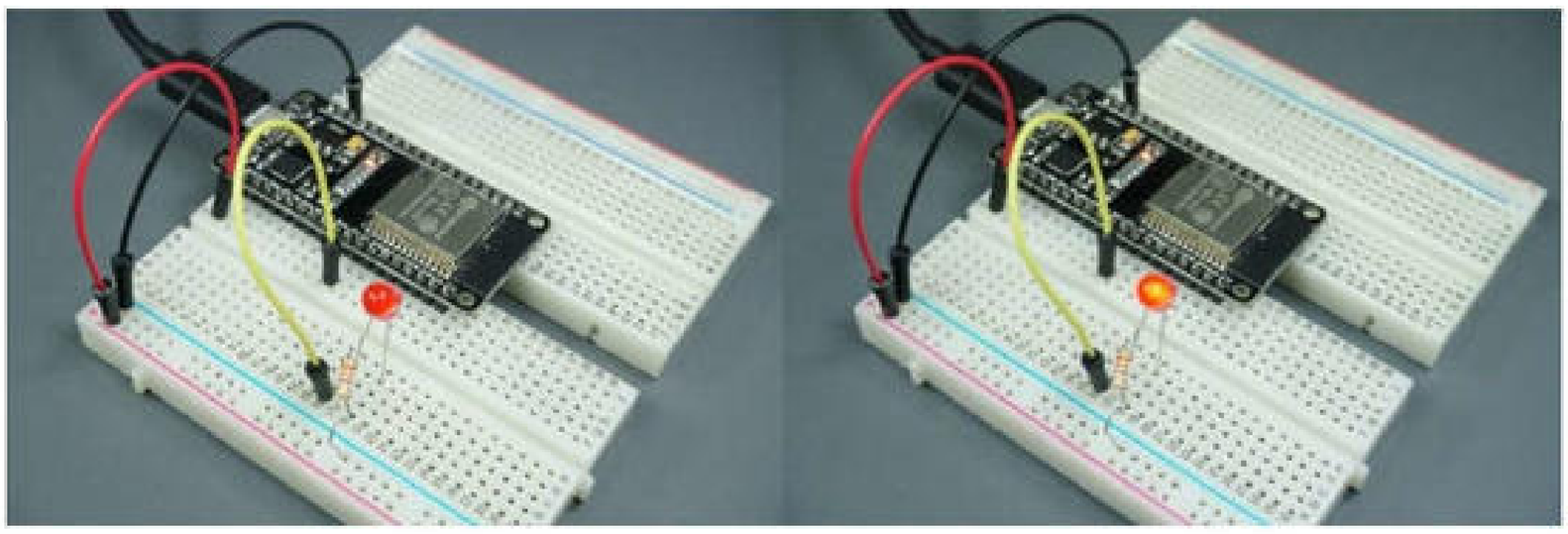
```
led.duty(duty_cycle)
```

After that, the `duty_cycle` is incremented by 1. The following table helps you understand what's happening in the code.

	duty_cycle	led.duty(duty_cycle)	GPIO state
loop 1	0	led.duty(0)	0% brightness
loop 2	1	led.value(1)	~ 0.1% brightness
loop 3	2	led.value(2)	~ 0.2% brightness
...	...	...	...
loop 1024	1023	led.value(1023)	100% brightness

Demonstration

Upload the code to your ESP32 or ESP8266. The LED attached to GPIO 5 should increase brightness over time.



LED Dimmer Switch Project

We can apply the concepts learned previously to build an LED dimmer switch: control the LED brightness using a potentiometer.

Schematic

For this project you need to wire a potentiometer to GPIO 34 (ESP32) or ADC 0 (ESP8266) and an LED to GPIO 5.

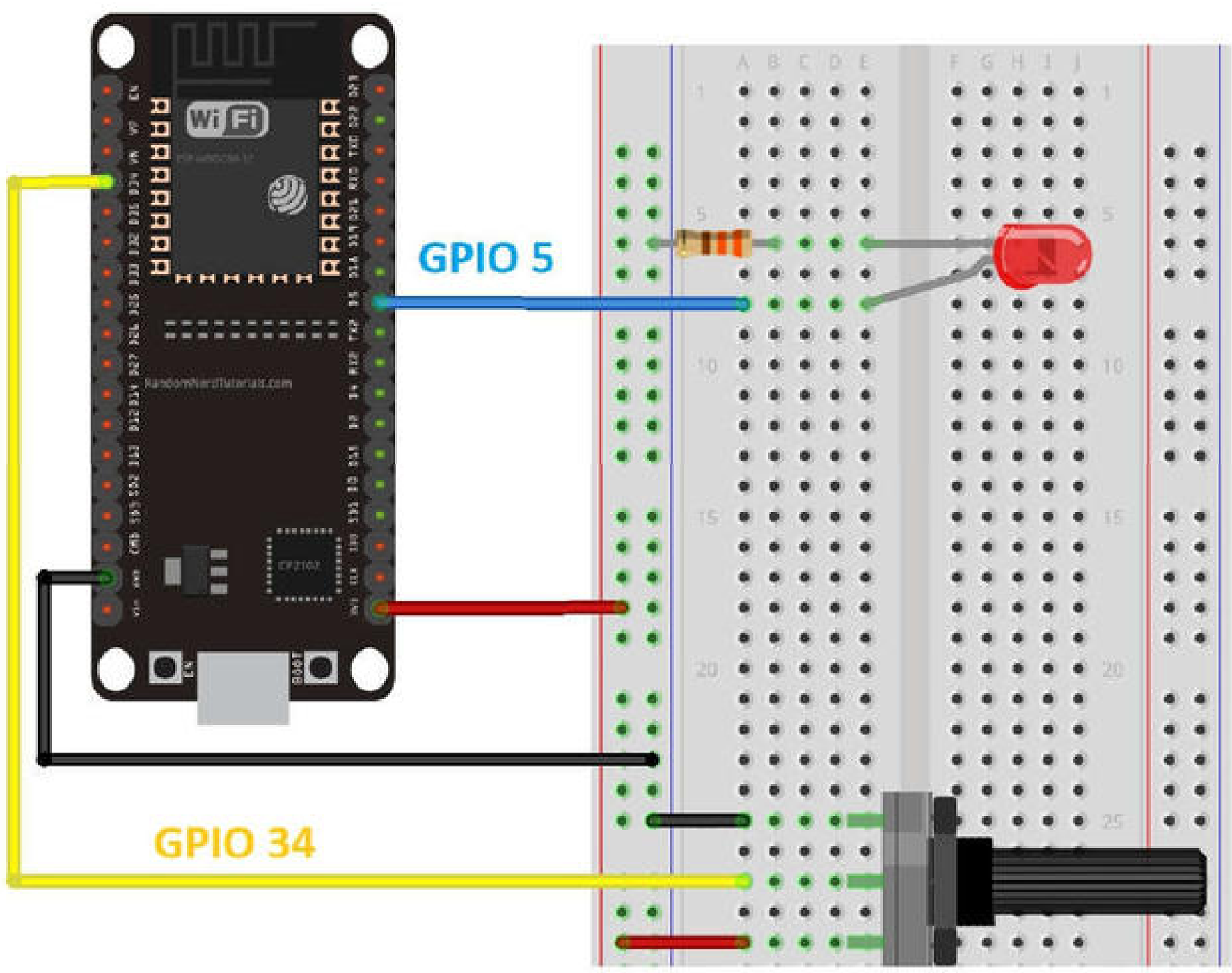
Parts required

Here's a list of parts you need to build the circuit:

- ESP32 or ESP8266
- 5mm LED
- 330 Ohm resistor
- Potentiometer
- Breadboard
- Jumper wires

Schematic – ESP32

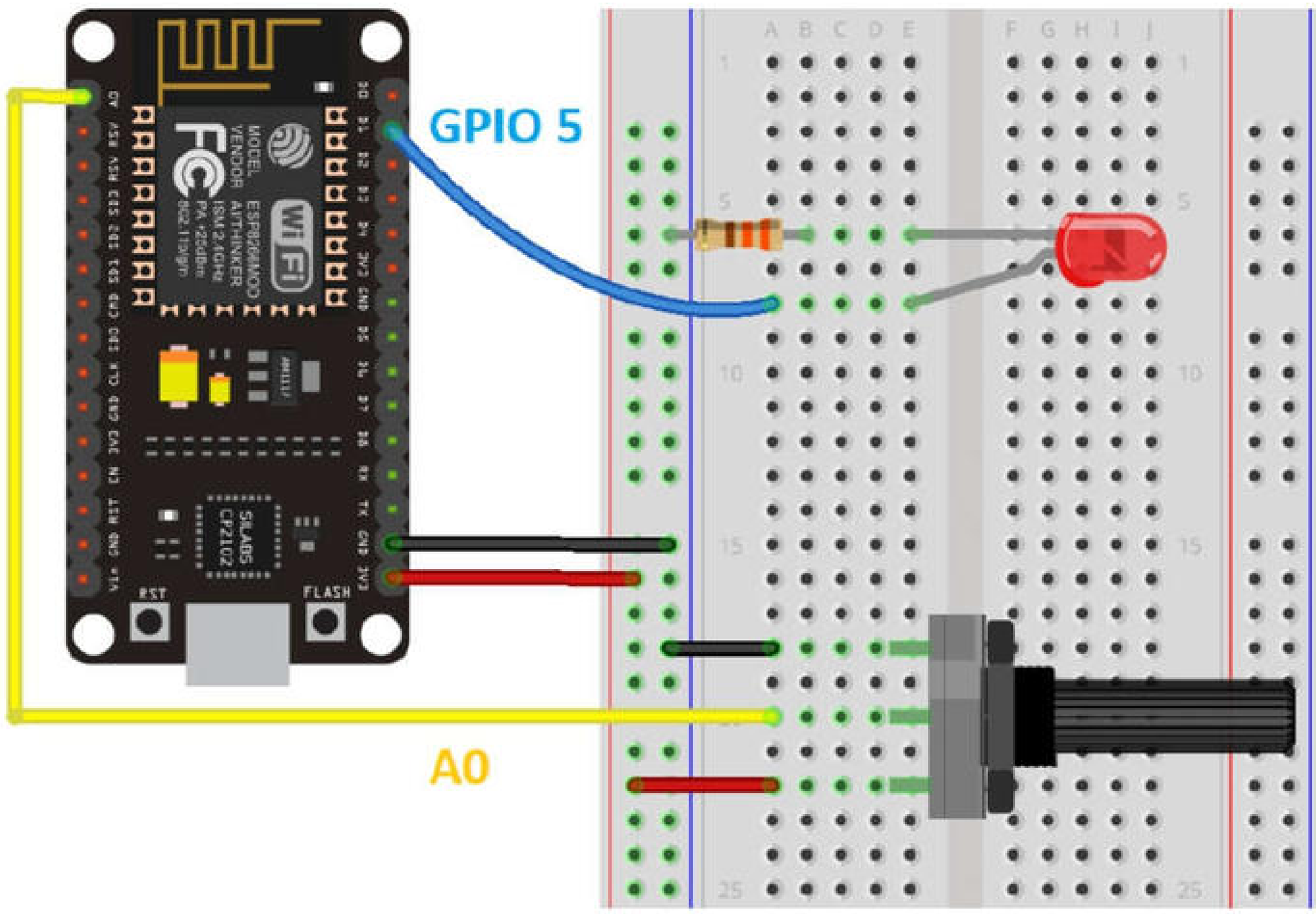
Follow the next schematic diagram if you're using an ESP32 board:



(This schematic uses the ESP32 DEVKIT DOIT board version with 36 GPIOs. Before assembling the circuit double-check the pinout for the board you're using.)

Schematic – ESP82

Follow the next schematic diagram if you're using an ESP8266 board:



Script

Because ADC works differently in ESP32 and ESP8266, we have a different script for each board.

Script - ESP32

Copy the following code to the *main.py* if you're using an ESP32 board.

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

frequency = 5000
led = PWM(Pin(5), frequency)
pot = ADC(Pin(34))
pot.width(ADC.WIDTH_10BIT)
pot.atten(ADC.ATTN_11DB)

while True:
  pot_value = pot.read()
  print(pot_value)
```

```
if pot_value < 15:
  led.duty(0)
else:
  led.duty(pot_value)

sleep(0.1)
```

SOURCE CODE
https://github.com/RuiSantosdotme/ESP-MicroPython/blob/master/code/GPIOs/PWM_Pot/pwm_pot_esp32.py

How the code works

As we've seen previously, we need to import the `Pin`, `PWM`, and `ADC` class from the `machine` module, and the `sleep` class from the `time` module.

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

Define the frequency of the PWM signal and create a `PWM` object called `led` on GPIO 5.

```
frequency = 5000
led = PWM(Pin(5), frequency)
```

Define an `ADC` object on pin 34, the GPIO the potentiometer is connected to.

```
pot = ADC(Pin(34))
```

In this specific example, we'll set the analog reading to 10-bit resolution. This simplifies our code, because this way we have the same range of values (0-1023) for both the analog reading, and the duty cycle.

```
pot.width(ADC.WIDTH_10BIT)
```

We want to be able to read full range (0-3.3V), so we need to add the following line:

```
pot.atten(ADC.ATTN_11DB)
```

In the `while` loop, we read the `pot` value, save it in the `pot_value` variable and print the value on the Shell:

```
pot_value = pot.read()
print(pot_value)
```

Then we have a condition that ensures the LED turns off with small potentiometer readings. If the `pot` value is smaller than 15, we set the duty cycle to 0.

```
if pot_value < 15:
  led.duty(0)
```

As seen previously, to set the duty cycle we simply need to use the *duty()* method on a `PWM` object, in this case the `led` variable.

```
led.duty(pot_value)
```

If the potentiometer reading is more than 15, we'll set the reading from the potentiometer as the duty cycle:

```
else:
  led.duty(pot_value)
```

Script – ESP8266

Copy the following code to the *main.py* if you're using an ESP8266 board.

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

frequency = 5000
led = PWM(Pin(5), frequency)
pot = ADC(0)

while True:
  pot_value = pot.read()
  print(pot_value)

  if pot_value < 15:
    led.duty(0)
  else:
    led.duty(pot_value)

  sleep(0.1)
```

SOURCE CODE
https://github.com/RuiSantosdotme/ESP-MicroPython/blob/master/code/GPIOs/PWM_Pot/pwm_pot_esp8266.py

How the code works

As we've seen previously, we import the `Pin`, `PWM`, and `ADC` class from the `machine` module, and the `sleep` class from the `time` module.

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

Define the frequency of the PWM signal, and create a `PWM` object called `led` on GPIO 5:

```
frequency = 5000
led = PWM(Pin(5), frequency)
```

Define an `ADC` object on the ESP8266 analog pin (A0).

```
pot = ADC(0)
```

In the `while` loop, we read the pot value, save it in the `pot_value` variable and print the value on the Shell:

```
pot_value = pot.read()
print(pot_value)
```

We have a condition that ensures the LED turns off with small potentiometer readings. If the pot value is smaller than 15, we set the duty cycle to 0.

```
if pot_value < 15:
  led.duty(0)
```

As explained previously, to set the duty cycle, use the *duty()* method on a `PWM` object, in this case the `led` variable.

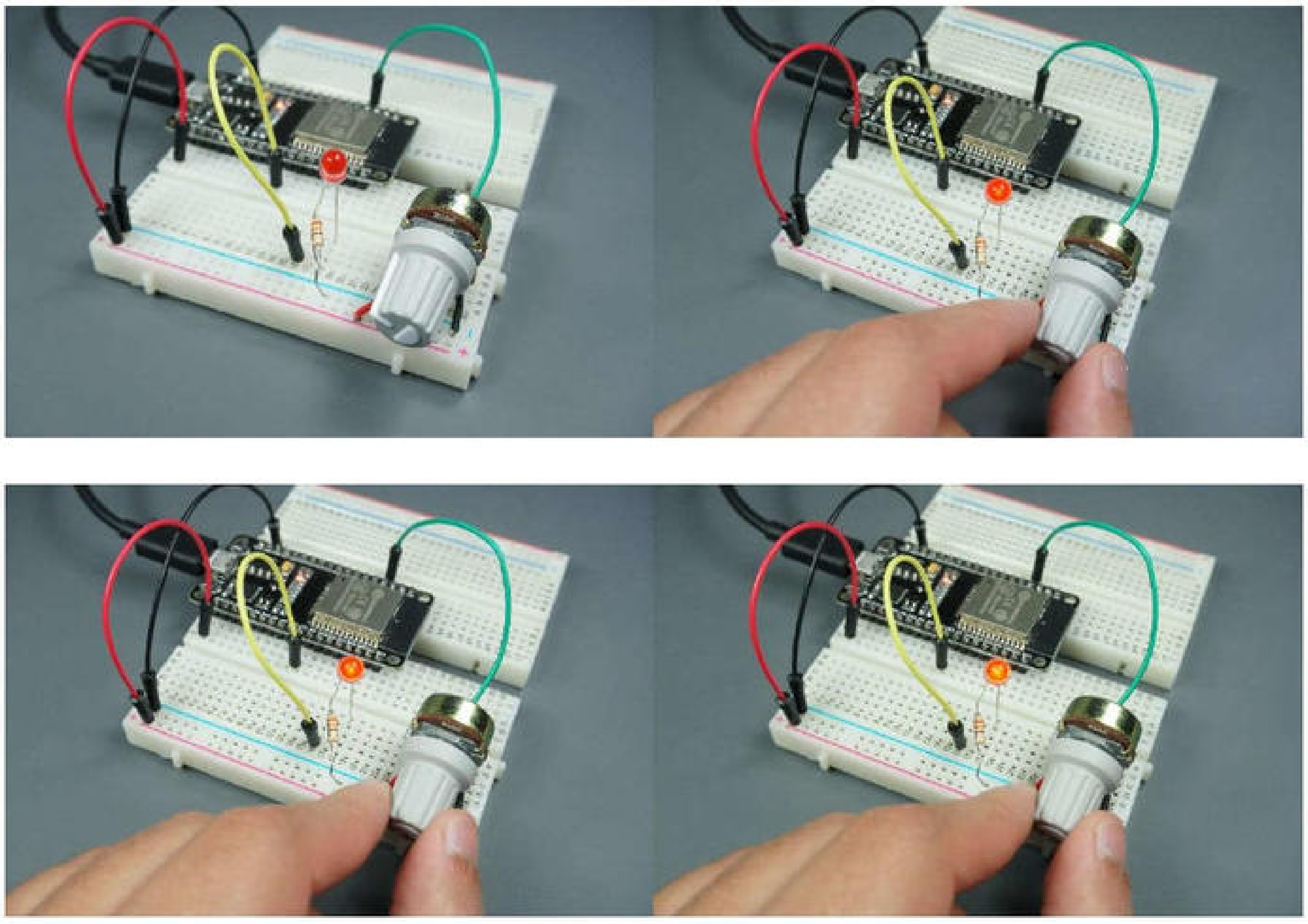
```
led.duty(pot_value)
```

If the potentiometer reading isn't smaller than 15, we'll set the reading from the potentiometer as the duty cycle:

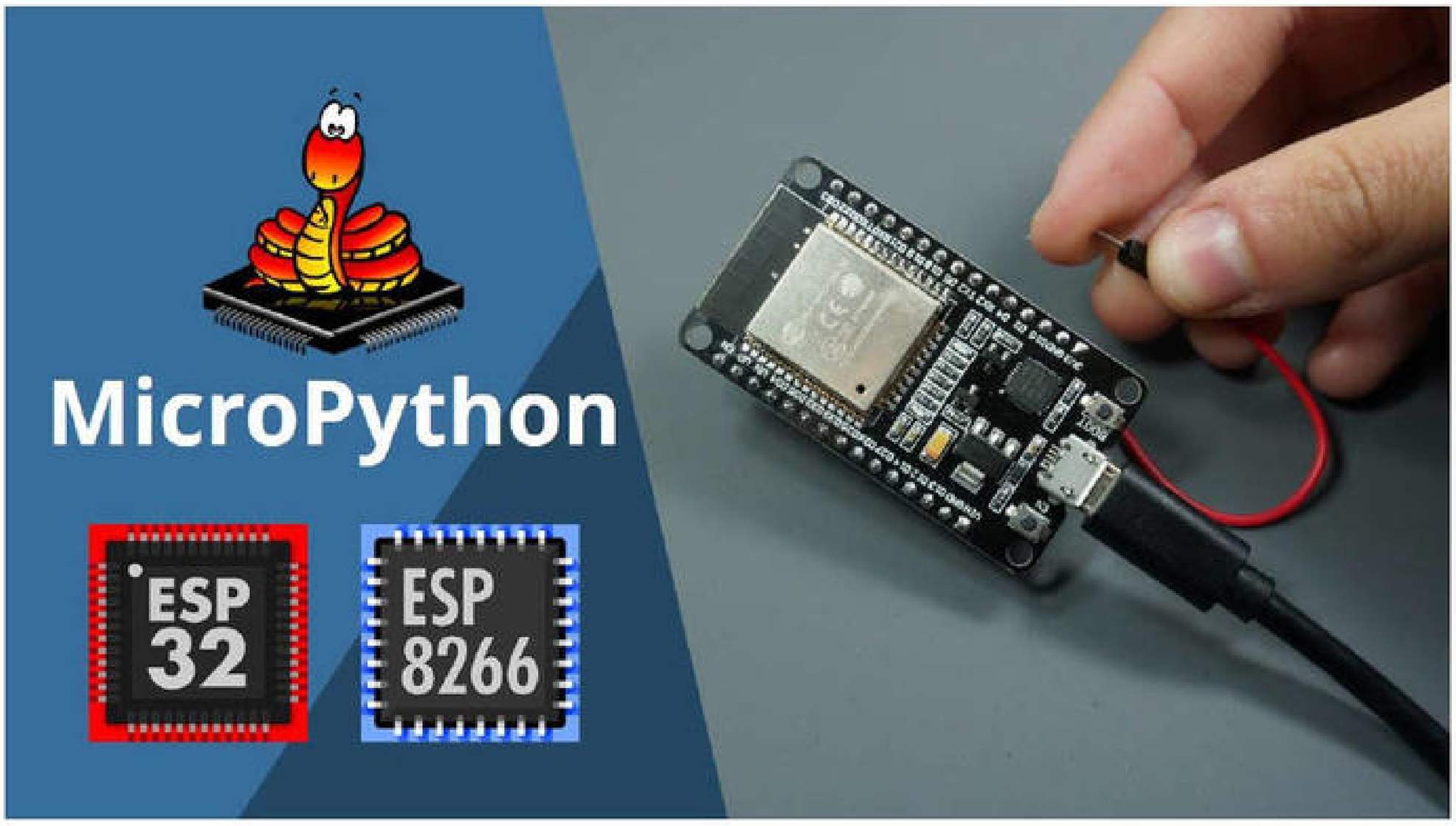
```
else:
  led.duty(pot_value)
```

Demonstration

Upload the code to your ESP32 or ESP8266 board. Then, you should be able to control the LED brightness by rotating the potentiometer. The readings from the potentiometer are printed on the Shell.



ESP32 Touch Pins



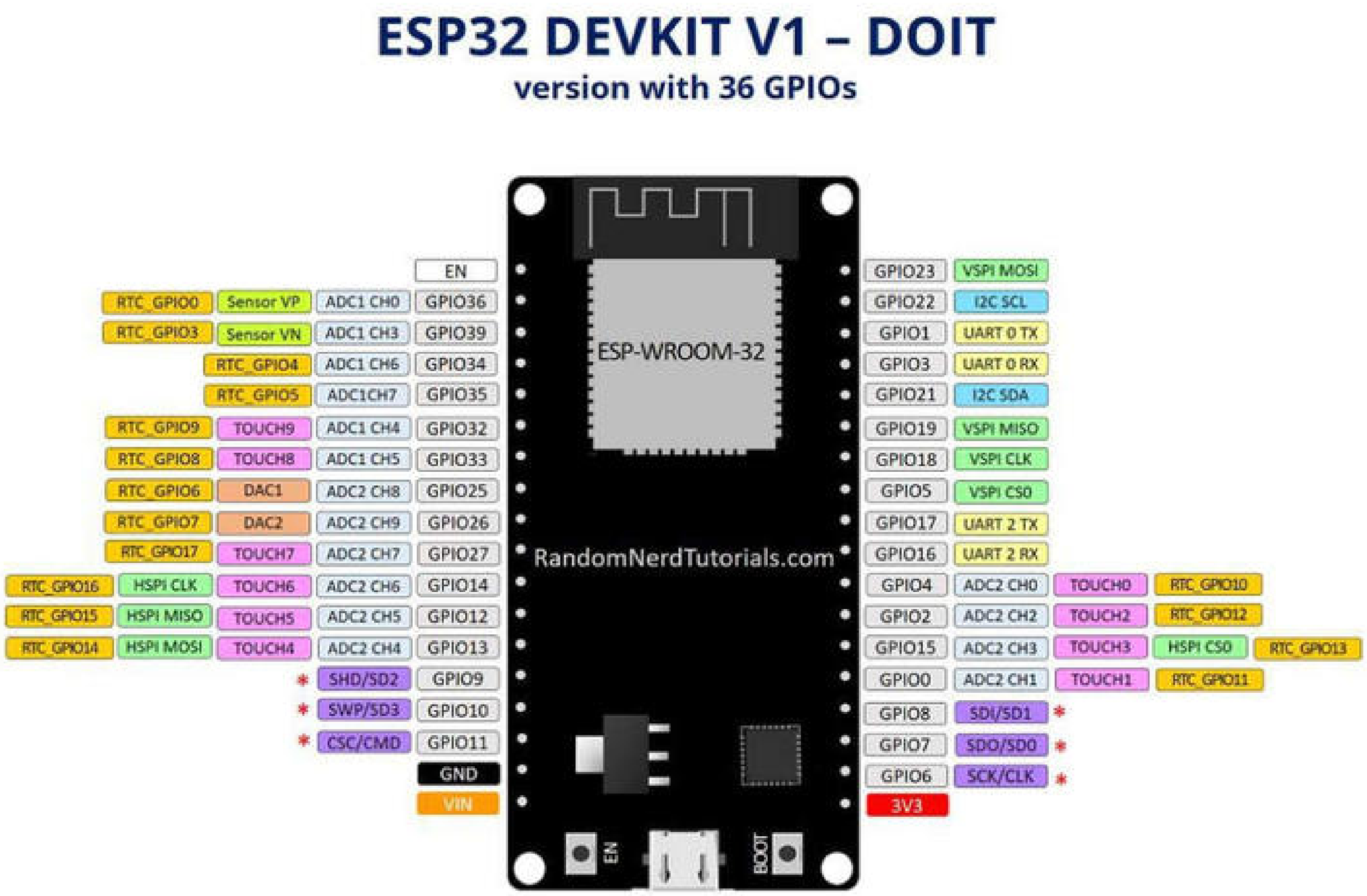
This section is only applicable to the ESP32 – it's about the ESP32 touch sensor.

Introducing the ESP32 Touch Sensor

The ESP32 has 10 capacitive sensing GPIOs. These GPIOs can sense variations in anything that holds an electrical charge, like the human skin. So, they can detect variations induced when touching the GPIOs with a finger. Here are the capacitive touch GPIOs:

- T0 (GPIO 4)
- T1 (GPIO 0)
- T2 (GPIO 2)
- T3 (GPIO 15)
- T4 (GPIO 13)
- T5 (GPIO 12)
- T6 (GPIO 14)
- T7 (GPIO 27)
- T8 (GPIO 33)
- T9 (GPIO 32)

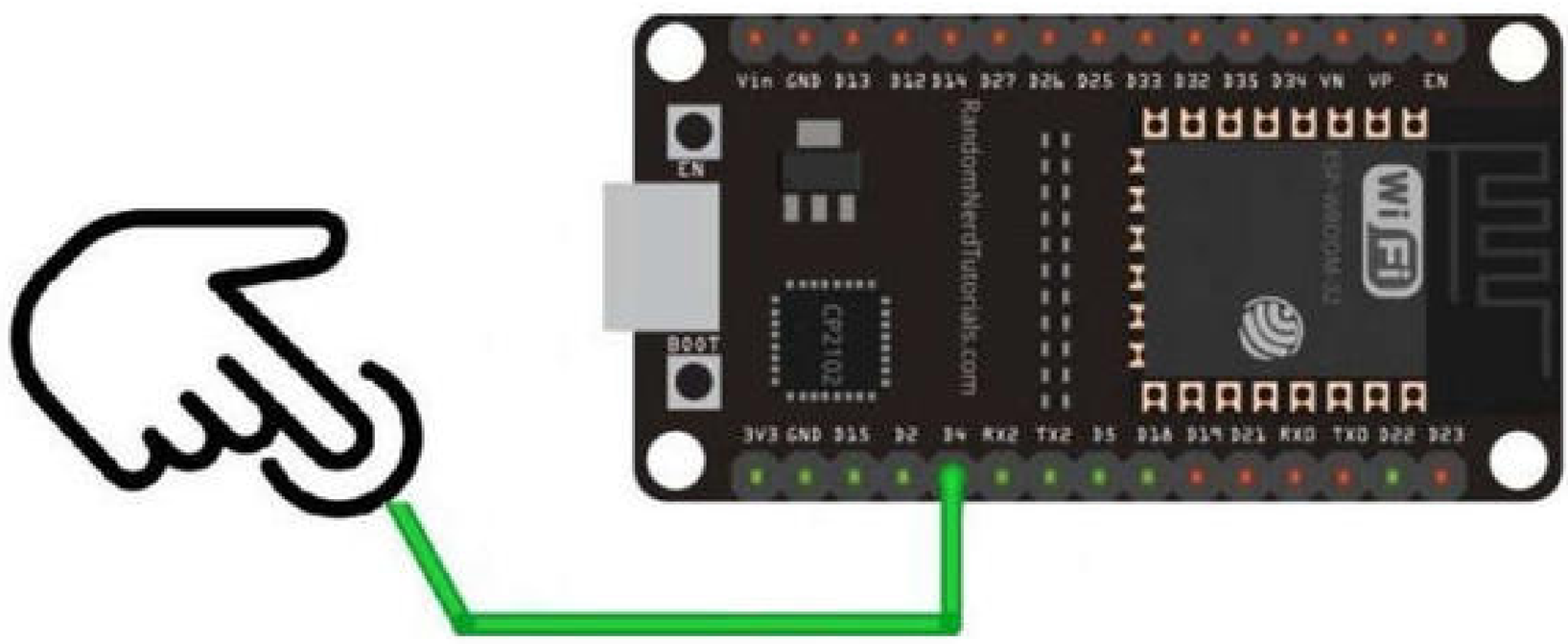
These are highlighted with a pink color on the next diagram.



Note: Touch sensor 1 is GPIO 0. However, it's not available as a pin in the ESP32 model with 30 GPIOs. GPIO 0 is available on the board with 36 pins.

Schematic

In this example we'll show you how to read the touch sensor value from a touch pin. We'll read Touch Pin 0 that corresponds to GPIO 4. Wire a cable to GPIO 4, in a way that you can touch the metal part:



Script

The following script simply reads the touch value from GPIO 4 every half second and prints the results on the Shell.

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

touch_pin = TouchPad(Pin(4))
while True:
  touch_value = touch_pin.read()
  print(touch_value)
  sleep(0.5)
```

SOURCE CODE
https://github.com/RuiSantosdotme/ESP-MicroPython/blob/master/code/GPIOs/ESP32_Touch_Pin/main_esp32.py

To read from the ESP32 touch pins, you need to import the `TouchPad` class from the `machine` module. You also need the `Pin` class to setup the GPIOs, and the `sleep()` method to add delays.

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

Then, create a `TouchPad` object called `touch_pin` on GPIO 4:

```
touch_pin = TouchPad(Pin(4))
```

Finally, to read the value from the touch pin, use the `read()` method on the `TouchPad` object as follows:

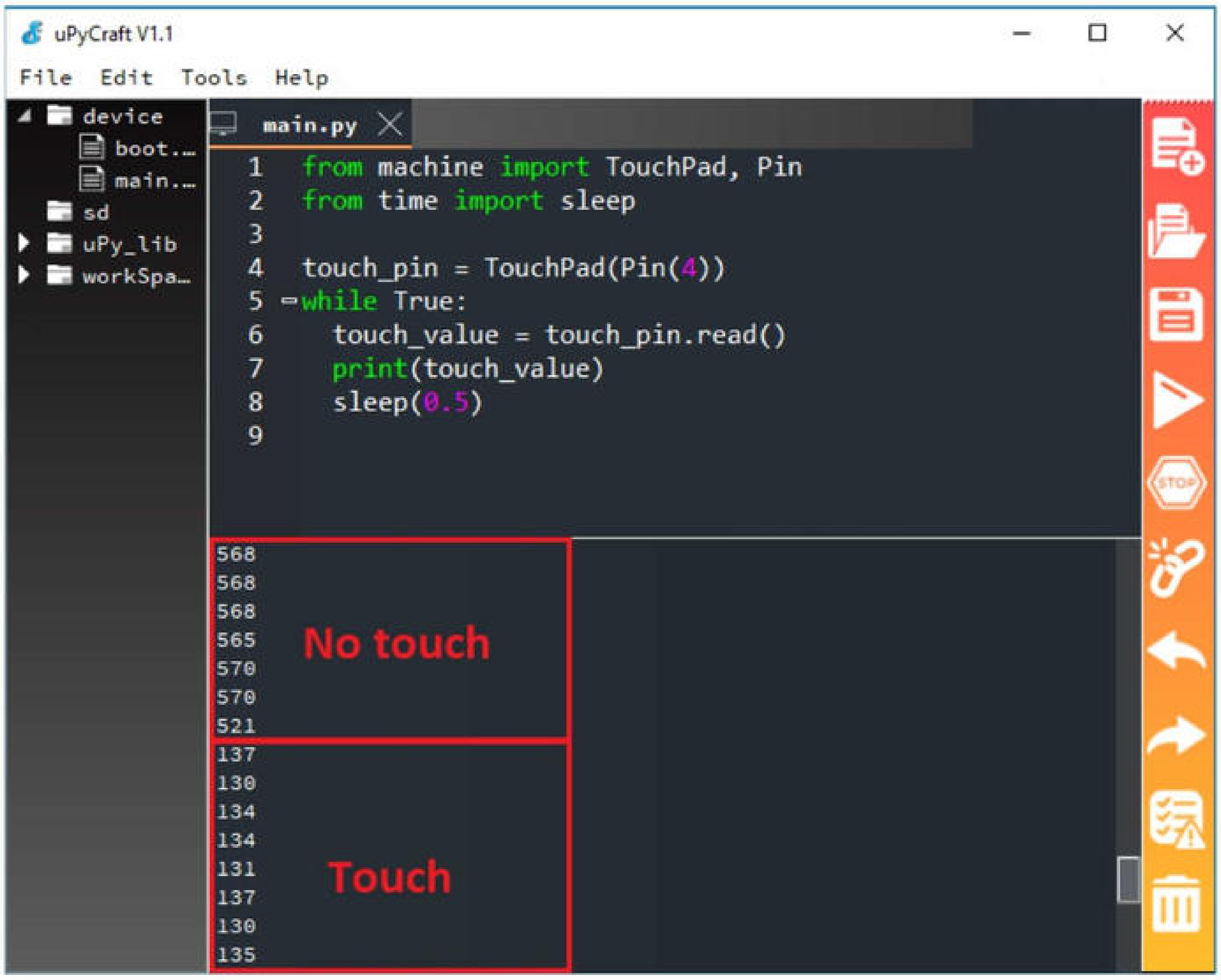
```
touch_value = touch_pin.read()
```

Then, the touch pin reading is printed on the Shell:

```
print(touch_value)
```

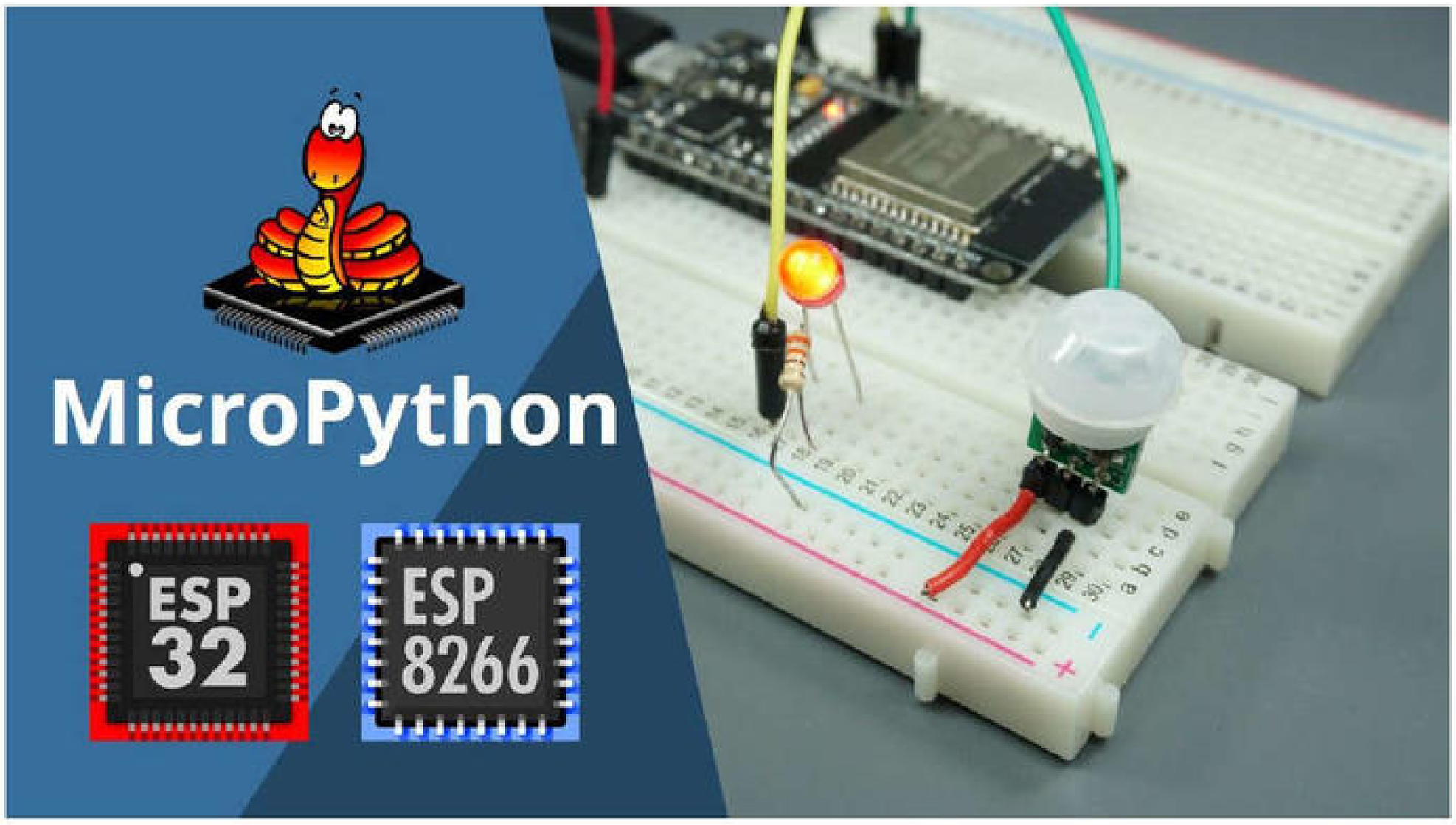
Demonstration

Copy the previous script to your *main.py* file and upload it to the ESP32 board. Then, touch the wire on GPIO 4 and see the values changing. You'll get lower values printed when you touch the pin with your finger.



Note: the values you get depend on the board, on your surroundings, on the wires you're using and other external factors.

Interrupts

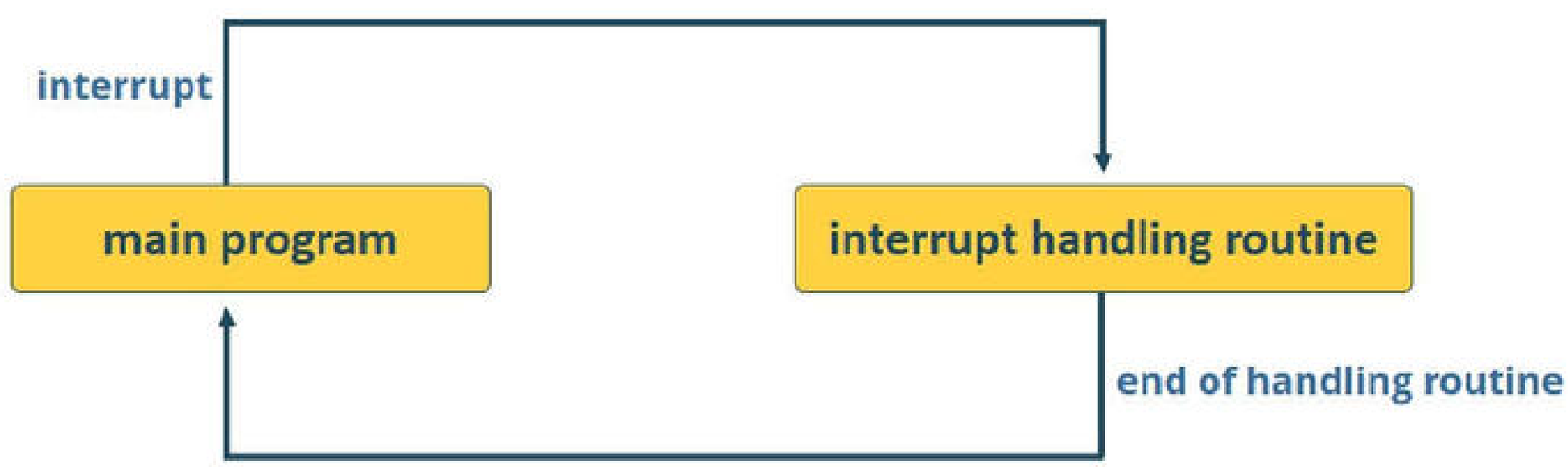


In this Unit, we'll show you how to handle interrupts in MicroPython and build a project example with a PIR motion sensor.

Introducing Interrupts

Interrupts are useful for making things happen automatically in microcontroller programs and can help solve timing problems.

With interrupts you don't need to constantly check the current pin value. When a change is detected, an event is triggered (a function is called). When an interrupt happens, the processor stops the execution of the main program to execute a task, and then gets back to the main program as shown in the figure below.



This is especially useful to trigger an action whenever motion is detected or whenever a pushbutton is pressed without the need for constantly checking its state.

ESP32 interrupt pins: all GPIOs, except GPIO 6 to GPIO 11.

ESP8266 interrupt pins: all GPIOs, except GPIO 16.

Set Up an Interrupt in MicroPython

To setup an interrupt in MicroPython, you need to follow the next steps:

1. Define an interrupt handling function.

The interrupt handling function should be as simple as possible, so the processor gets back to the execution of the main program quickly. The best approach is to signal the main code that the interrupt has happened by using a global variable, for example. The interrupt handling function should accept a parameter of type `Pin`. This parameter is returned to the callback function and it refers to the GPIO that caused the interrupt.

```
def handle_interrupt(pin):
```

2. Setup the GPIO that will act as an interrupt pin as an input. For example:

```
pir = Pin(14, Pin.IN)
```

3. Attach an interrupt to that pin by calling the `irq()` method:

```
pir.irq(trigger=Pin.IRQ_RISING, handler=handle_interrupt)
```

The `irq()` method accepts the following arguments:

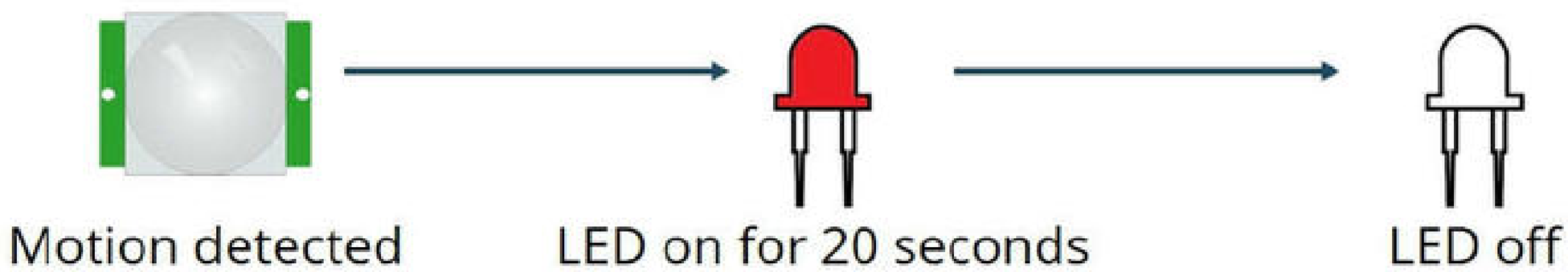
- **trigger**: this defines the trigger mode. There are 3 different conditions:
  - **`Pin.IRQ_FALLING`**: to trigger the interrupt whenever the pin goes from HIGH to LOW;
  - **`Pin.IRQ_RISING`**: to trigger the interrupt whenever the pin goes from LOW to HIGH.

- 3: to trigger the interrupt in both edges (this means, when any change is detected)

- **handler**: this is a function that will be called when an interrupt is detected, in this case the `handle_interrupt()` function.

Project Example with PIR Motion Sensor

To demonstrate how to handle interrupts, we'll build a simple project with a PIR motion sensor. Whenever motion is detected we'll light up an LED for 20 seconds.



Schematic

For this project you need to wire a PIR motion sensor and an LED to your ESP32 or ESP8266 board.

> To use interrupts with the ESP32, you can use all GPIOs, except GPIO 6 to GPIO 11.
>
> With the ESP8266, you can use interrupts in all GPIOs, except GPIO 16.

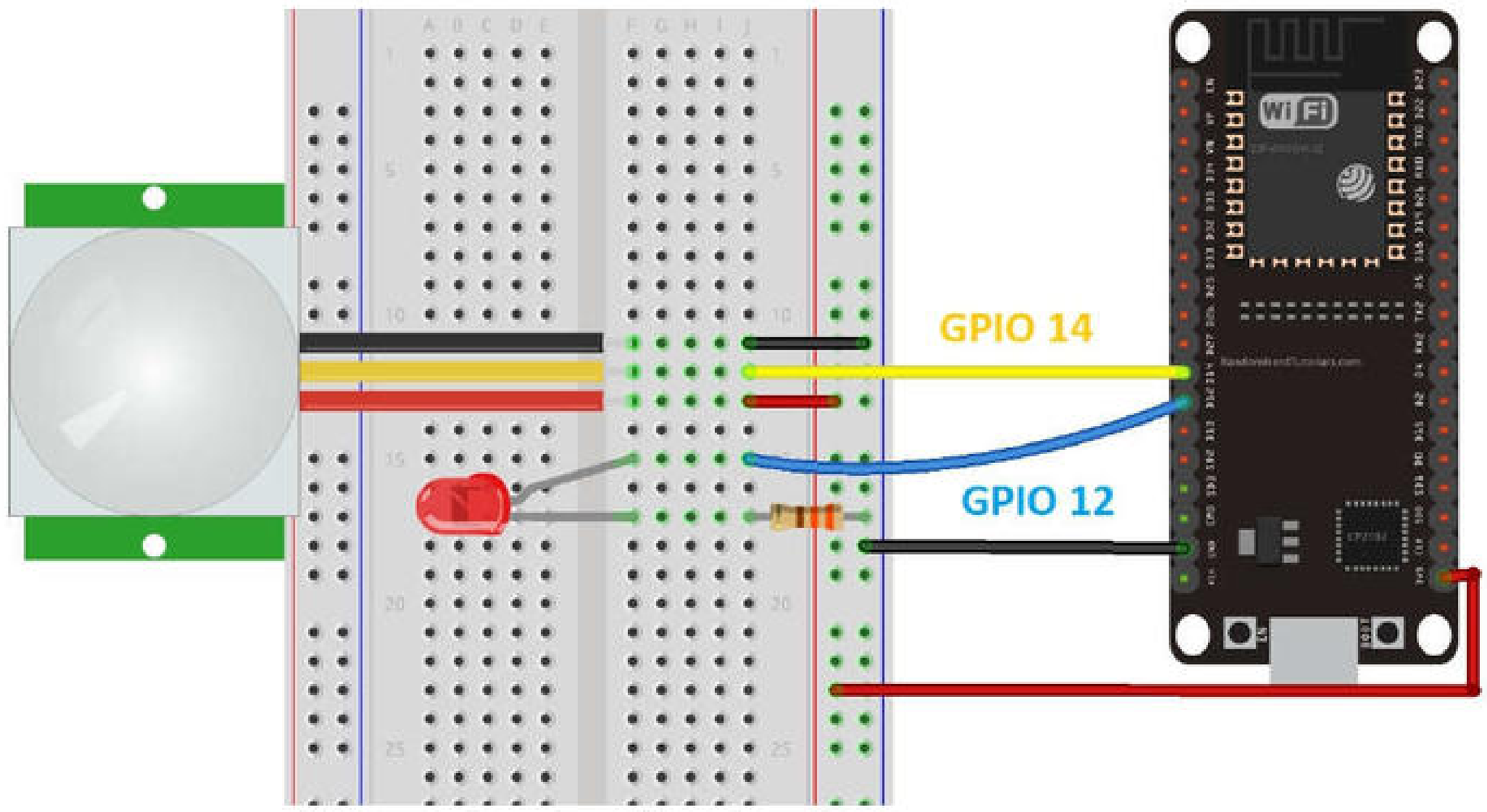
Parts required

Here's a list of parts you need to build the circuit:

- ESP32 or ESP8266
- 5mm LED
- 330 Ohm resistor
- Mini PIR motion sensor (AM312) or PIR motion sensor (HC-SR501)
- Breadboard
- Jumper wires

Schematic – ESP32

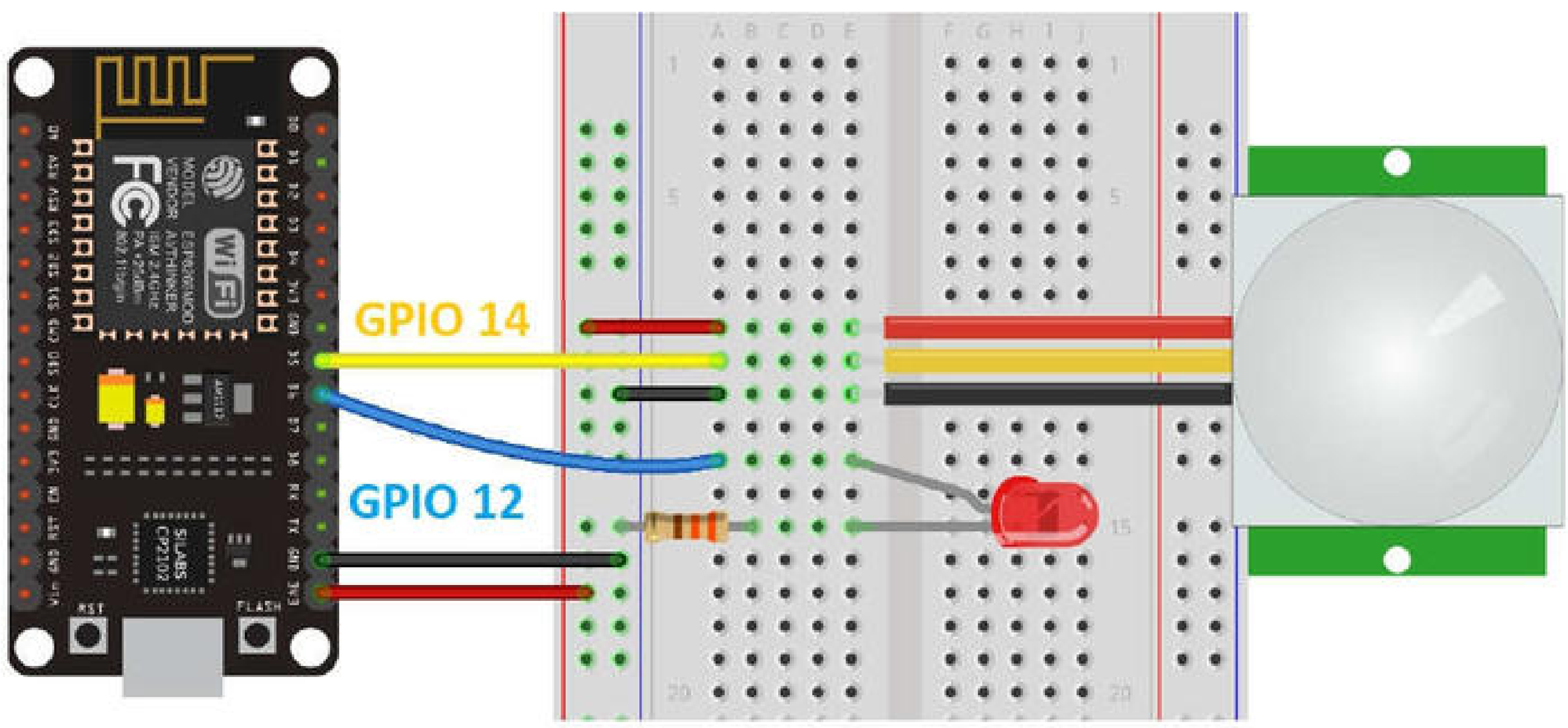
Follow the next schematic diagram if you're using an ESP32 board:



This schematic uses the ESP32 DEVKIT DOIT board version with 36 GPIOs. Before assembling the circuit double-check the pinout for the board you're using.

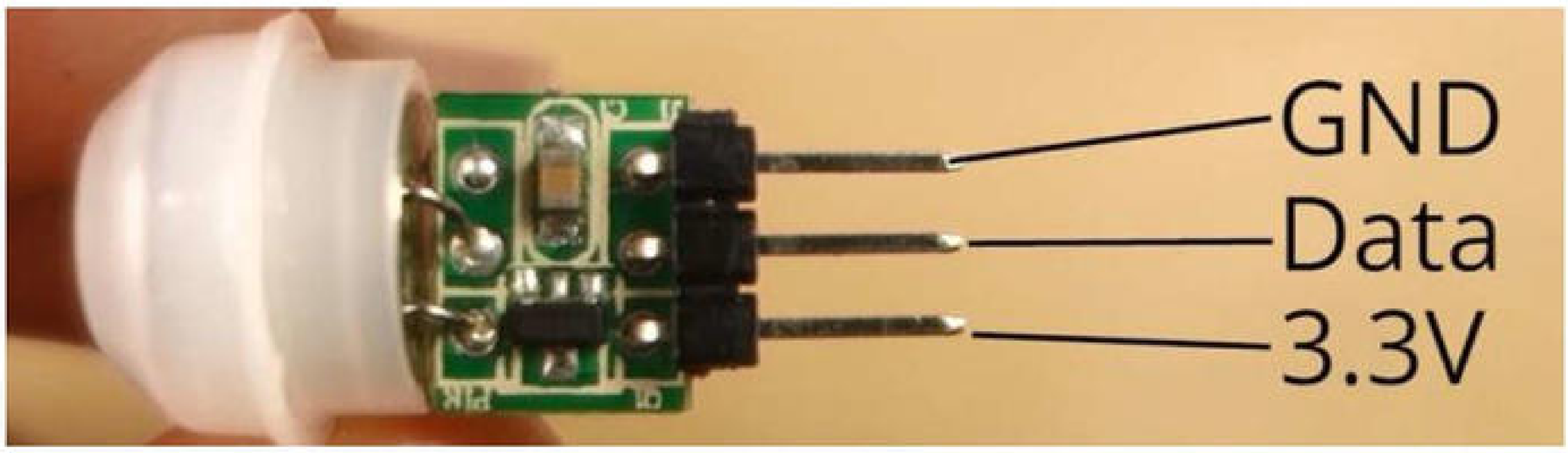
Schematic – ESP8266

Follow the next schematic diagram if you're using an ESP8266 board:



Note: the Mini AM312 PIR Motion Sensor we're using in this project operates at 3.3V. However, if you're using another PIR motion sensor like the HC-SR501, it operates at 5V. You can either modify it to operate at 3.3V or simply power it using the Vin pin.

In the figure below, we provide the pinout for the Mini AM312 PIR motion sensor. If you're using another motion sensor, please check its pinout before assembling the circuit.



Script

Here's the script that detects motion and lights up an LED whenever motion is detected. This code is compatible with both the ESP32 and ESP8266.

```
from machine import Pin
from time import sleep

motion = False

def handle_interrupt(pin):
  global motion
  motion = True
  global interrupt_pin
  interrupt_pin = pin

led = Pin(12, Pin.OUT)
pir = Pin(14, Pin.IN)

pir.irq(trigger=Pin.IRQ_RISING, handler=handle_interrupt)

while True:
  if motion:
    print('Motion detected! Interrupt caused by:', interrupt_pin)
    led.value(1)
    sleep(20)
    led.value(0)
    print('Motion stopped!')
    motion = False
```

SOURCE CODE
https://github.com/RuiSantosdotme/ESP-MicroPython/blob/master/code/GPIOs/PIR_Interrupt_Delay/pir_interrupt_delay.py

How the Code Works

To use interrupts, we import the `Pin` class from the `machine` module. We also import the `sleep` method from the `time` module to add a delay in our script.

```
from machine import Pin
from time import sleep
```

Start by creating a variable called `motion` that can be either `True` or `False`. This variable will indicate whether motion was detected or not (this is the global variable that will be changed on the interrupt handling function).

```
motion = False
```

Then, create a function called `handle_interrupt`.

```
def handle_interrupt(pin):
  global motion
  motion = True
  global interrupt_pin
  interrupt_pin = pin
```

This function will be called every time motion is detected. The `handle_interrupt` function has an input parameter (`pin`) in which an object of class `Pin` will be passed when the interrupt happens.

Here we're saving the pin that caused the interrupt in the `interrupt_pin` variable. In this case, it is not very useful because we only have one interrupt pin. However, this can be useful if we have several interrupts that trigger the same interrupt handling function and we want to know which GPIO caused the interrupt.

In our example, the `handle_interrupt` function simply changes the `motion` variable to `True` and saves the interrupt pin.

Important: `motion` is defined as a **global** variable, because we want it to be accessible throughout all the code, not just inside of the `handle_interrupt` function.

Variable's scope: A *variable's scope* is the area of a program where a variable is accessible. In Python, there are two basic variable scopes: ***local*** and ***global***.

A variable defined in the main code body is global, meaning it is accessible anywhere else in the code. A variable defined inside a function is local to that function, so what you do with the local variable inside the function has no effect on variables outside, even if they have the same name.

As you want `motion` to be usable both inside the function and throughout the code, it needs to be declared as **global**. Otherwise, when motion is detected nothing would happen, because the `motion` variable would be changing inside the function and not in the main body of the code.

Interrupts are handled in interrupt service routines (ISR), which is the function that runs when the interrupt happens. When an interrupt happens, the processor stops what it is doing, works temporarily on the ISR, and then gets back to what it was doing before (the main code).

It is a good practice to build ISR as small as possible, so the processor gets back to the execution of the main program as soon as possible. The best approach is to signal the main code that the interrupt has happened by using a flag or a counter, for example. Here, we use the `motion` variable as a flag to indicate that motion was detected. Then, the main code should have all the things we want to happen when motion is detected like turning an LED on.

Let's proceed with the code. We need to create two `Pin` objects. One for the LED on GPIO 12, and another for the PIR motion sensor on GPIO 14. Because we'll read the signal from the motion sensor, it is an input, so we use `Pin.IN`.

```
led = Pin(12, Pin.OUT)
pir = Pin(14, Pin.IN)
```

Then, set an interrupt on the `pir` by calling the `irq()` method.

```
pir.irq(trigger=Pin.IRQ_RISING, handler=handle_interrupt)
```

In the loop, when the `motion` variable is True, we turn the LED on for 20 seconds and print a message that indicates that motion was detected, and which pin caused the interrupt.

```
if motion:
  print('Motion detected! Interrupt caused by:', interrupt_pin)
  led.value(1)
  sleep(20)
```

After 20 seconds, turn the LED off, and print a message "Motion stopped!".

```
led.value(0)
print('Motion stopped!')
```

Finally, set the `motion` variable to `False`:

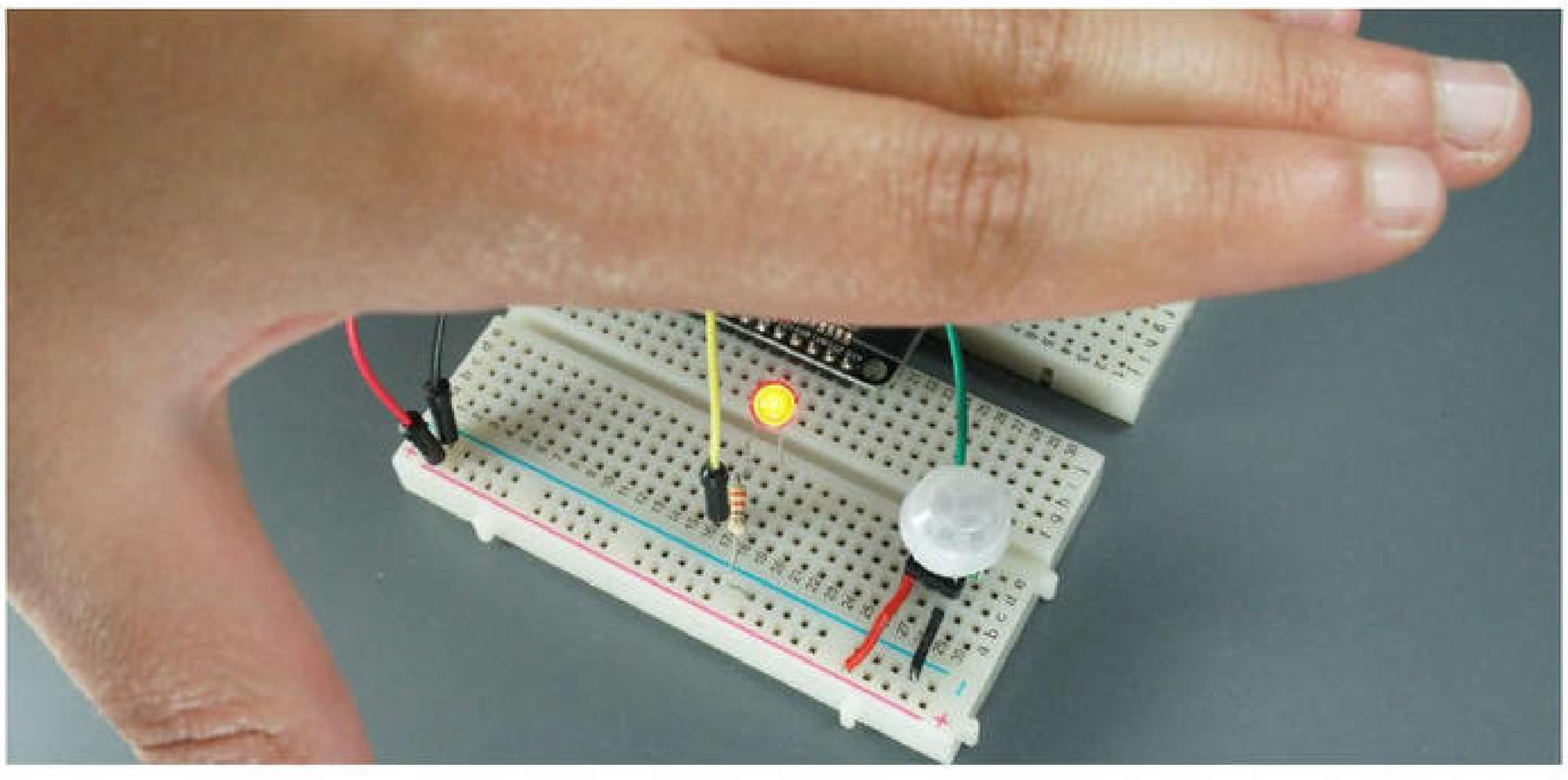
```
motion = False
```

The `motion` variable can only become `True` again, if motion is detected and the `handle_interrupt` function is called.

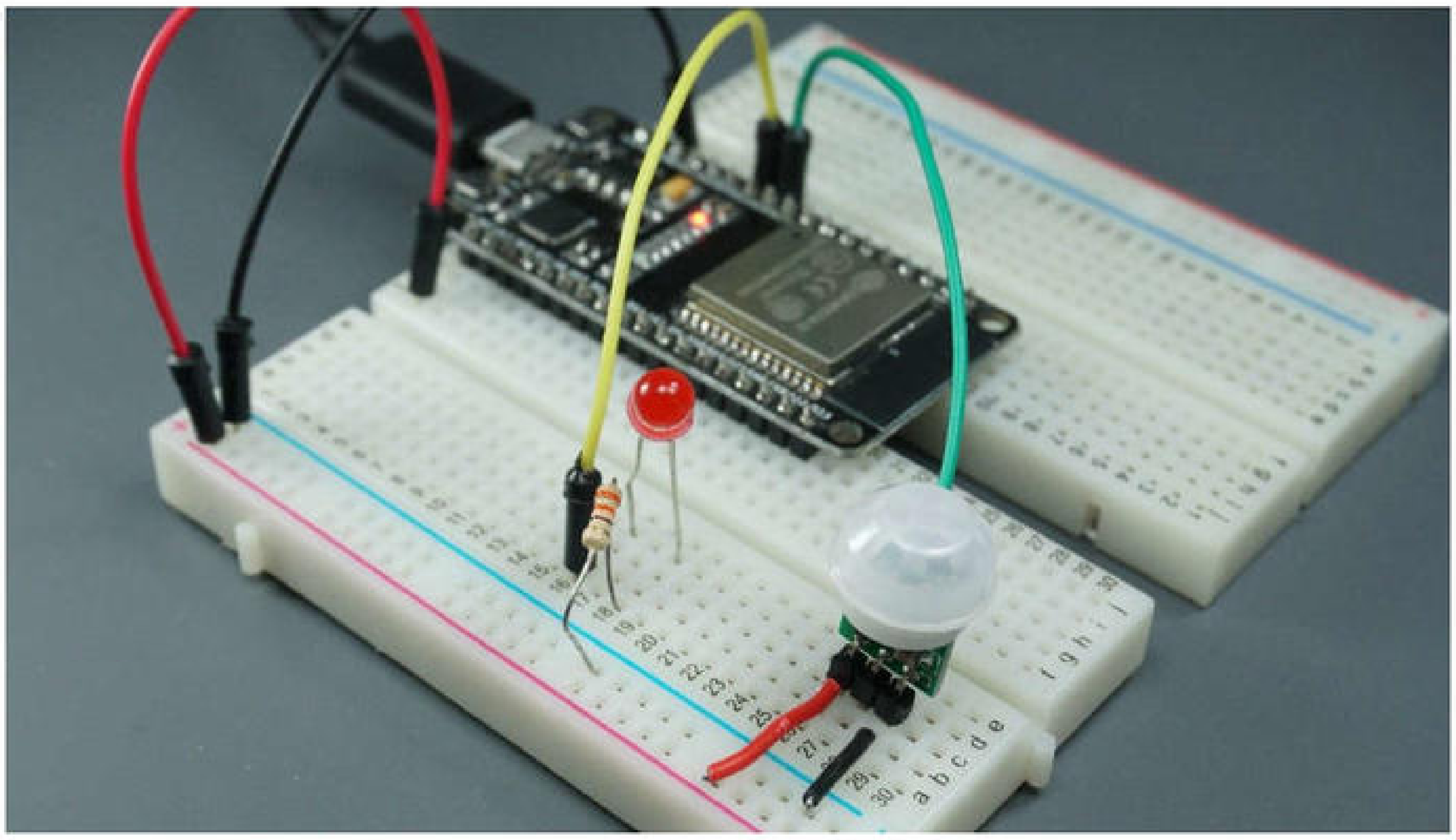
For simplicity, in this example we use a delay to keep the LED on for 20 seconds. Ideally, you should use timers (see the next unit).

Demonstration

Upload the code to your ESP32/ESP8266 board. The LED should turn on for 20 seconds when motion is detected, and a message should be printed in the Shell.

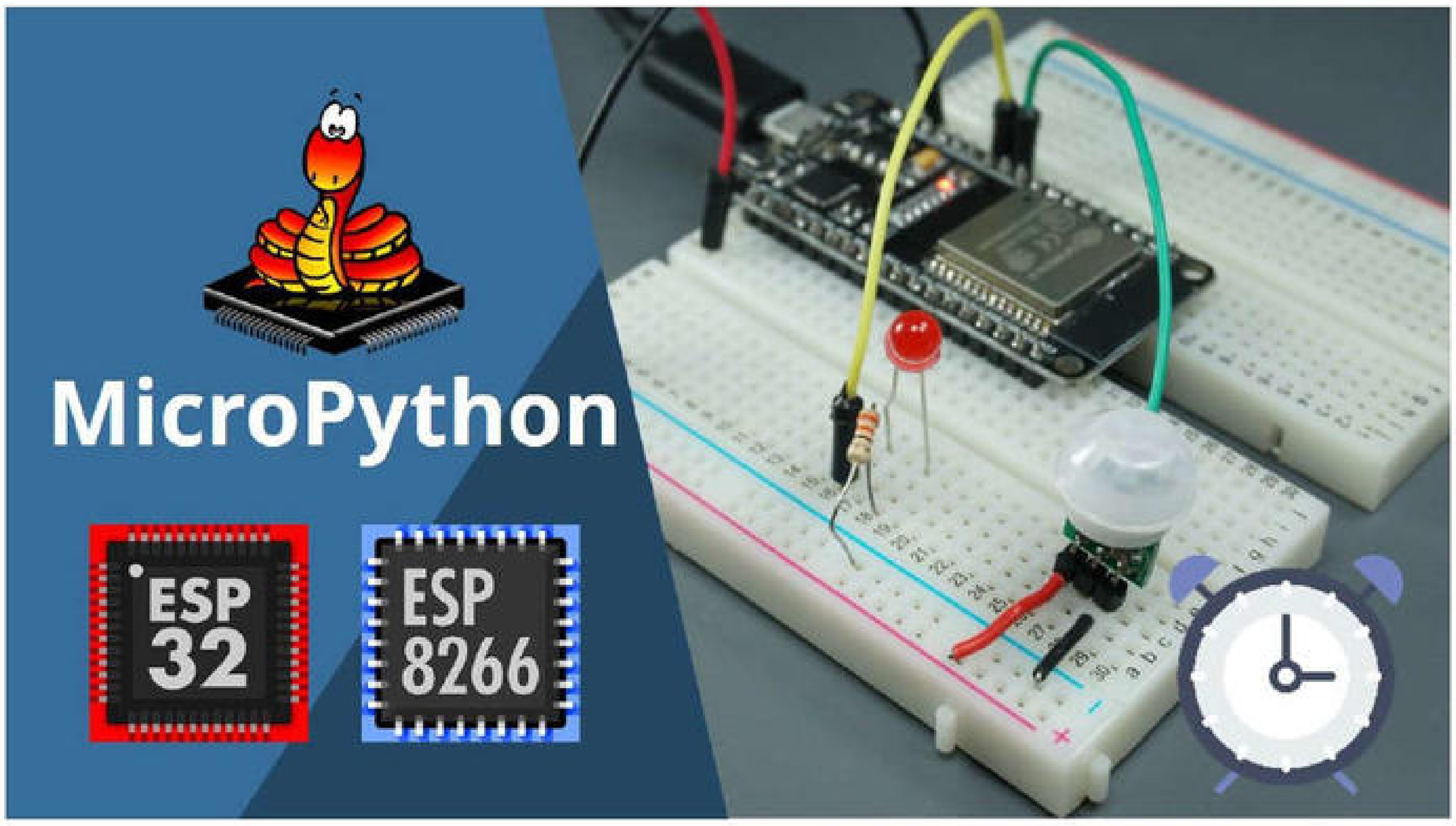


After 20 seconds the LED turns off.



Note: the AM312 PIR motion sensor has a default delay time of 8 seconds. This means that it won't be triggered before 8 seconds have passed since the last trigger.

Timers



In this Unit, we'll introduce you to timers. We'll build the same project of the previous section but using timers.

The *sleep()* method

Until now, we've used the `sleep()` method that is pretty straightforward. It accepts a number as an argument. This number represents the time in seconds the program waits until moving on to the next line of code.

The `sleep()` method is a blocking function. Blocking functions prevent a program from doing anything else until that task is completed. If you need multiple tasks to occur at the same time, you cannot use `sleep()`. For example, in the previous project, you can't do anything else while the LED is on.

Instead of using the `sleep()` method that blocks your code and doesn't allow you to do anything else for a determined number of seconds, we'll use a timer. For most projects you should avoid using delays and use timers instead.

The *time()* method

There is a method from the `time` module that allows us to know how much time has passed since the program started. The `time()` method from the `time` module returns the number of seconds that have passed since the program started running.

Script

The following script illustrates how to use the `time()` method. It prints the time that has passed since the program started running on the ESP32 or ESP8266.

```
from time import time,sleep
while True:
  current_time = time()
  print(current_time)
  sleep(1)
```

SOURCE CODE
https://github.com/RuiSantosdotme/ESP-MicroPython/blob/master/code/GPIOs/Timer/timer_example.py

We import the `time` and the `sleep` methods.

```
from time import time,sleep
```

In the loop, save the current time on the `current_time` variable. To get the time that has passed since the program started running, we simply need to assign the `current_time` variable to the `time()` method.

```
current_time = time()
```

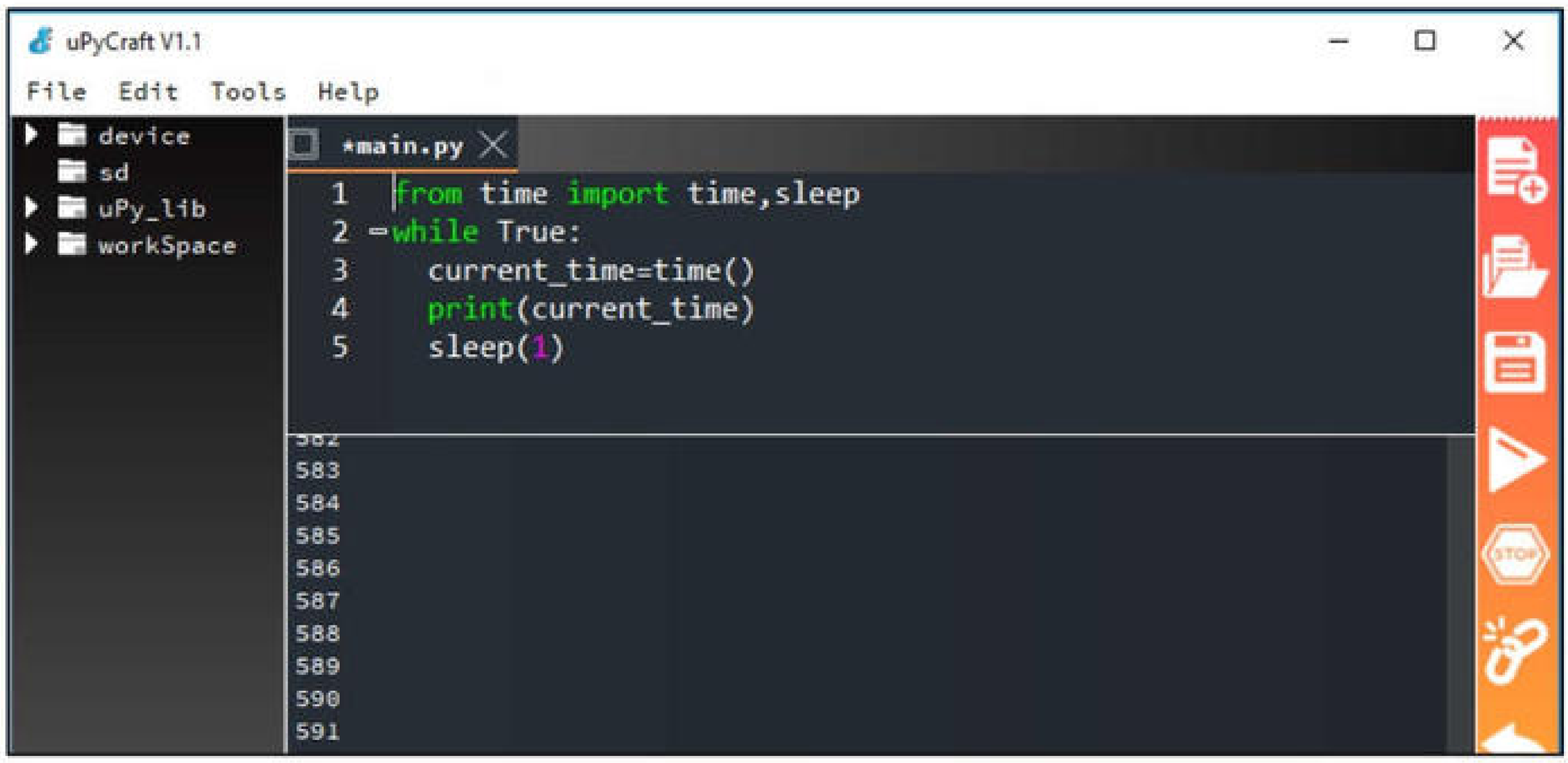
Then, print the time:

```
print(current_time)
```

The loop repeats every second.

```
sleep(1)
```

After uploading the code to the ESP32/ESP8266 and pressing the on-board EN button, you should see the numbers increasing every second.



PIR Motion Sensor with Interrupts and Timers

Here's an example on how to handle an interrupt from a PIR motion sensor with timers. Copy the following code to the *main.py* file of your ESP32/ESP8266 board (use the same schematic built in the previous Unit).

```
from machine import Pin
from time import time

start_timer = False
motion = False
last_motion_time = 0
delay_interval = 20

def handle_interrupt(pin):
  global motion, last_motion_time, start_timer
  motion = True
  start_timer = True
  last_motion_time = time()

led = Pin(12, Pin.OUT)
pir = Pin(14, Pin.IN)

pir.irq(trigger=Pin.IRQ_RISING, handler=handle_interrupt)

while True:
  if motion and start_timer:
    print('Motion detected!')
    led.value(1)
```

```
    start_timer = False

  elif motion and (time() - last_motion_time)>delay_interval:
    print('Motion stopped!')
    led.value(0)
    motion = False
```

SOURCE CODE
https://github.com/RuiSantosdotme/ESP-MicroPython/blob/master/code/GPIOs/PIR_Interrupt_Timer/pir_interrupt_timer.py

How the Code Works

First, import the `Pin` class from the `machine` module and the `time` method from the `time` module.

```
from machine import Pin
from time import time
```

We need to create some variables to keep track of what is going on in the code.

```
start_timer = False
motion = False
last_motion_time = 0
delay_interval = 20
```

The `start_timer` variable will tell us if the timer is running or not. The `motion` variable will hold if motion is detected. Finally, the `last_motion_time` saves the last time motion was detected.

After that, create the `handle_interrupt` function. This function runs every time motion is detected.

```
def handle_interrupt(pin):
  global motion, last_motion_time, start_timer
  motion = True
  start_timer = True
  last_motion_time = time()
```

When motion is detected, the `motion` variable changes to `True` and it starts the timer, so change the `start_timer` variable to `True`. Finally, save the current time in the `last_motion_time`, because this is the last time motion was detected.

```
last_motion_time = time()
```

After that, create two `Pin` objects, one for the LED and another for the PIR motion sensor.

```
led = Pin(12, Pin.OUT)
pir = Pin(14, Pin.IN)
```

We also need to set an interrupt on the GPIO the PIR is connected to:

```
pir.irq(trigger=Pin.IRQ_RISING, handler=handle_interrupt)
```

In the `while` loop is where the program will trigger an action depending on whether motion is detected and how much time has passed since the last time motion was detected.

First, we have an `if` statement that checks if motion was detected and if the timer has started:

```
if motion and start_timer:
```

If that condition is `True`, it means that motion was detected, so we turn the LED on and set the `start_timer` variable to `False`.

```
led.value(1)
start_timer = False
```

The next condition checks whether more than the 20 seconds (`delay_interval`) have passed since motion was detected.

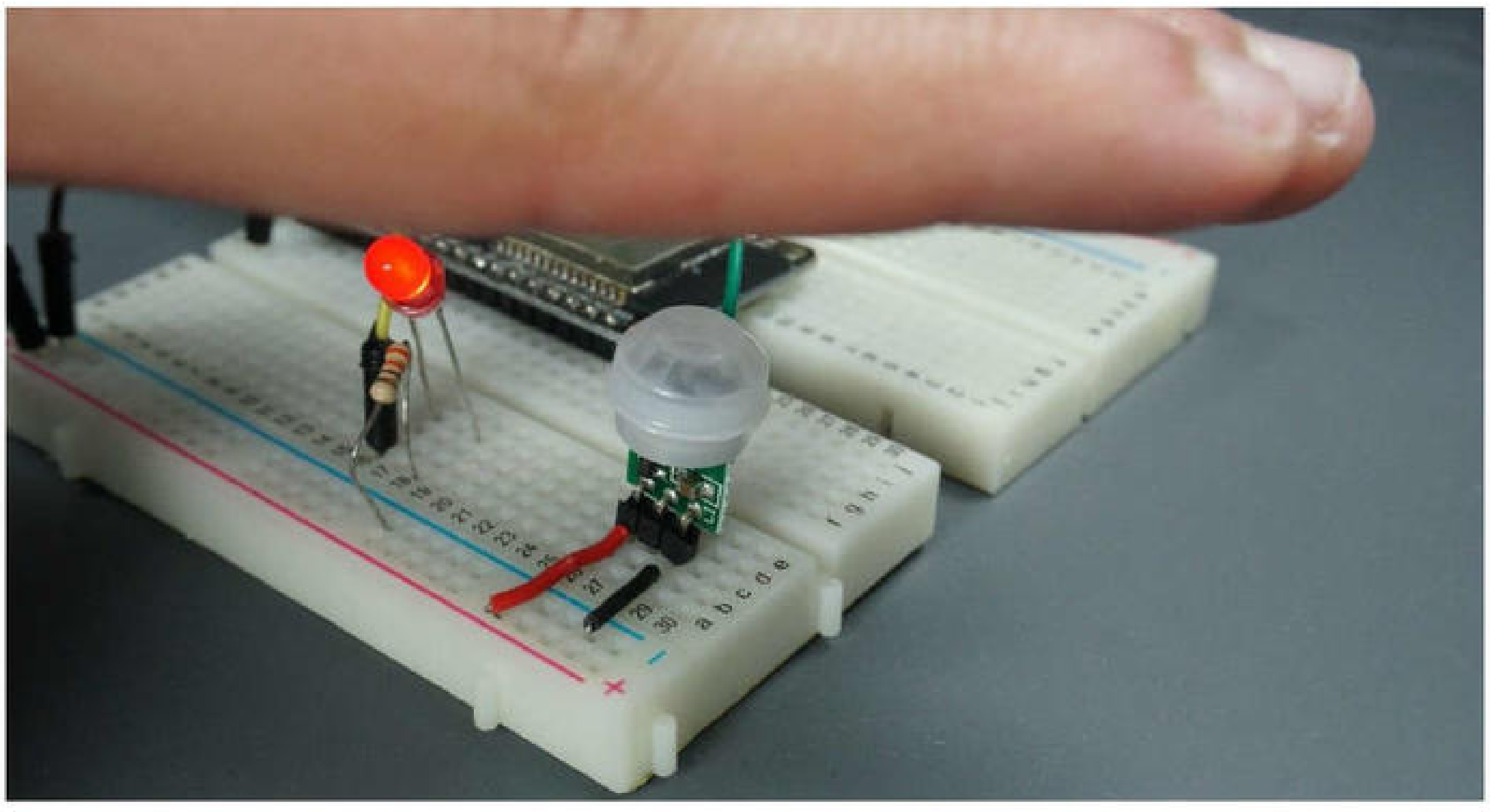
```
elif motion and (time() - last_motion_time) > delay_interval:
```

If more than 20 seconds have passed, we turn off the LED, and set the `motion` variable to `False`, so that we can detect motion again.

```
led.value(0)
motion = False
```

Demonstration

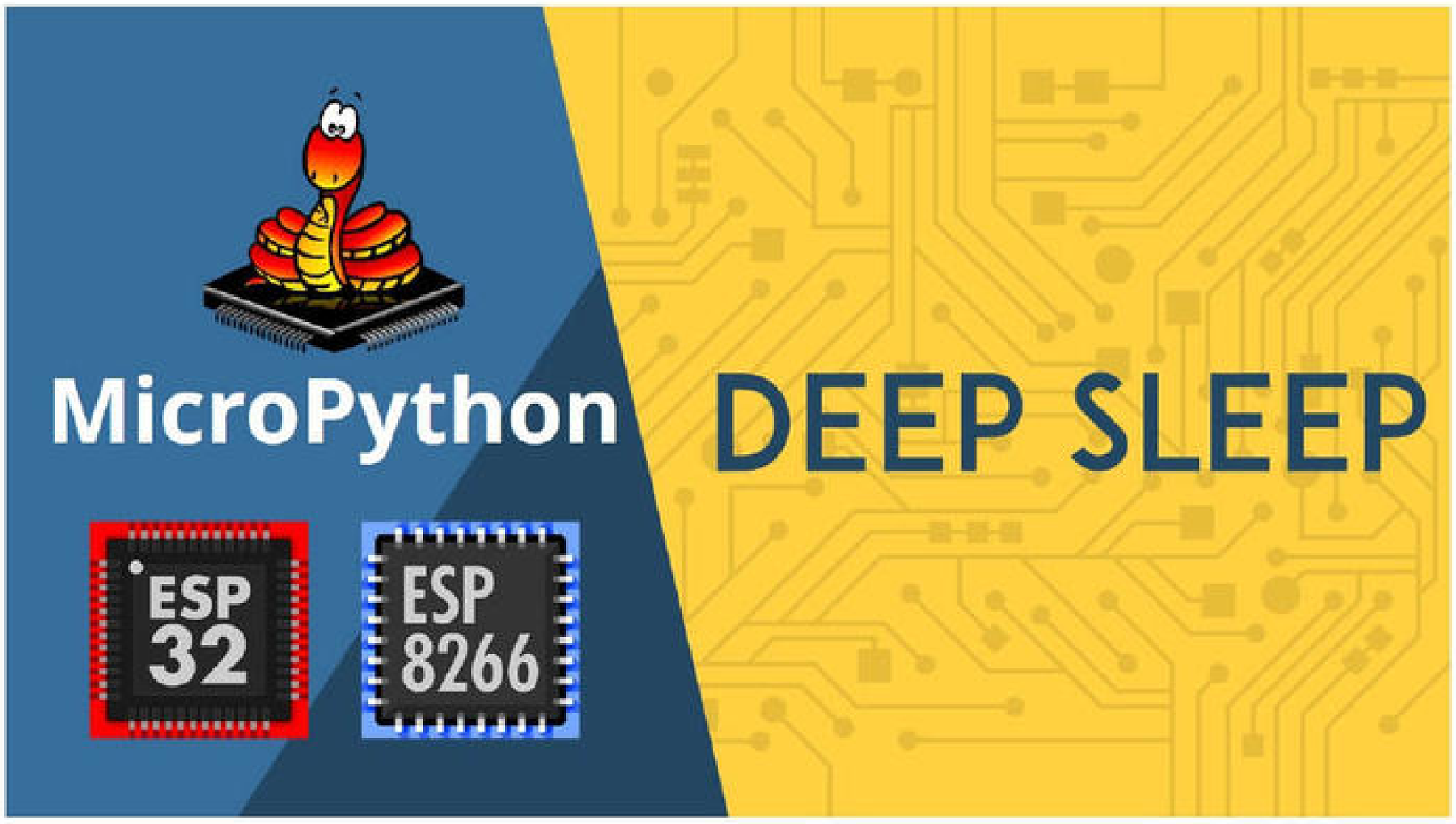
If you upload this code to your ESP board, you'll get the same result from the previous example.



However, you should understand that in this example your code doesn't get stuck waiting when motion is detected. Instead, you can add other tasks to be executed, and you'll still be able to detect motion and keep the LED on.

Note: the AM312 PIR motion sensor has a default delay time of 8 seconds. This means that it won't be triggered before 8 seconds have passed since the last trigger.

Deep Sleep with Timer Wake Up

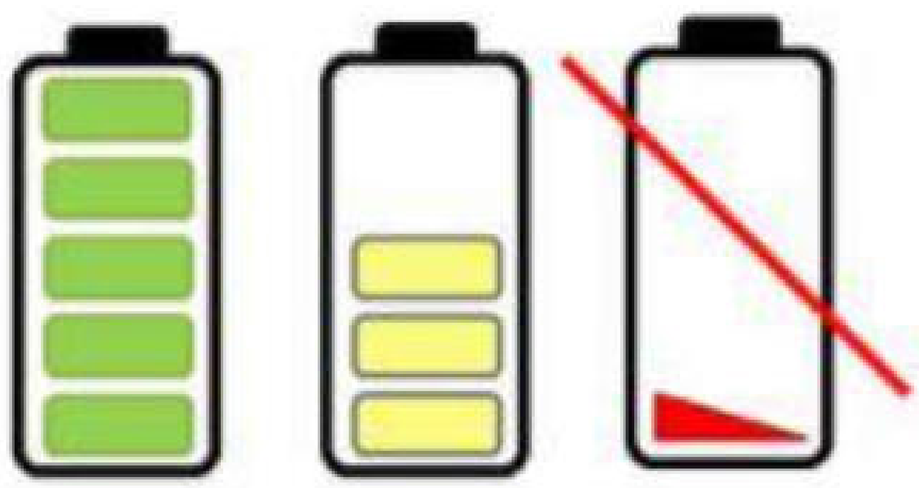


In this Unit, we'll show you how to put your ESP32 or ESP8266 in deep sleep mode and wake it up with a timer.

Note: at the time of writing this unit, not all deep sleep functions are implemented. For example, deep sleep with external wake up and touchpad are not yet fully implemented in the official MicroPython distribution. So, those topics won't be covered in this Unit.

Introducing Deep Sleep

Having your ESP32/ESP8266 running on active mode with batteries it's not ideal, since the power from batteries will drain very quickly.



If you put your ESP32/ESP8266 in deep sleep mode, it will reduce the power consumption and your batteries will last longer. Having the ESP32/ESP8266 in deep sleep mode means cutting with the activities that consume more power while operating but leave just enough activity to wake up the processor when something interesting happens.

When operating in deep sleep mode, the ESP32/ESP8266 have a current consumption on the µA range. However, if you use a full-feature board with built-in programmer, LEDs, and so on, you won't be able to achieve such a low power state. For example, the ESP32 has a deep sleep mode in which you can achieve with a custom and carefully designed board a minimal consumption of only 5 µA.

Wake Up Sources

After putting the ESP32 or ESP8266 into deep sleep mode, there are several ways to wake them up:

- You can use the timer: waking up your ESP32/ESP8266 after predefined periods of time;
- You can use an external wake up - this means the ESP can wake up when a change in the state of a pin occurs **– not yet implemented at the time of writing**;
- You can use the touch pins (only applies to ESP32) **– implemented, but not working as expected at the time of writing**;
- You can use the ULP co-processor to wake up (only applies to ESP32) **– not yet implemented at the time of writing**;

Throughout this Unit we'll cover timer wake up with both the ESP32 and ESP8266.

Timer Wake Up

Your ESP32 and ESP8266 can go into deep sleep mode, and then wake up at predefined periods of time. This feature is especially useful if you are running projects that require time stamping or daily tasks, while maintaining low power consumption.

Timer wake up works differently on ESP32 and ESP8266. Follow the right section depending on the board you're using.

ESP32 – Deep Sleep with Timer Wake Up

To put the ESP32 in deep sleep mode for a predetermined number of seconds, you just have to use the `deepsleep()` function from the `machine` module. This function accepts as arguments, the sleep time in milliseconds as follows:

```
machine.deepsleep(sleep_time_ms)
```

Let's look at a simple example to see how it works.

In the following code, the ESP32 is in deep sleep mode for 10 seconds, then it wakes up, blinks an LED, and goes back to sleep.

```
import machine
from machine import Pin
from time import sleep

led = Pin (2, Pin.OUT)

#blink LED
led.value(1)
sleep(1)
led.value(0)
sleep(1)

# wait 5 seconds so that you can catch the ESP awake to establish
a serial communication later
# you should remove this sleep line in your final script
sleep(5)

print('Im awake, but Im going to sleep')

# sleep for 10 seconds (10000 milliseconds)
machine.deepsleep(10000)
```

SOURCE CODE
https://github.com/RuiSantosdotme/ESP-MicroPython/blob/master/code/GPIOs/Deep_sleep/deep_sleep_esp32.py

First, import the necessary libraries:

```
import machine
from machine import Pin
from time import sleep
```

Create a `Pin` object that refers to GPIO 2 called `led`. This refers to the on-board LED.

```
led = Pin (2, Pin.OUT)
```

Then, the following lines blink the LED.

```
led.value(1)
sleep(1)
led.value(0)
sleep(1)
```

Before going to sleep, we add a delay of 5 seconds and print a message to indicate it's going to sleep.

```
sleep(5)
print('Im awake, but Im going to sleep')
```

It's important to add a 5 seconds delay before going to sleep when we are developing test scripts. When you want to upload a new code to the board, it needs to be awake. So, if you don't have the delay, it will be difficult to catch it awake to upload code. After having the final code, you can delete that delay.

Finally, put the ESP32 in deep sleep for 10 seconds (10 000 milliseconds).

```
machine.deepsleep(10000)
```

After 10 seconds, the ESP32 wakes up and runs the code from the beginning, similarly to when you press the EN/RST button.

Demonstration

Copy the code provided to the *boot.py* file of your ESP32. Upload the new code and press the **EN/RST** button after uploading. The ESP32 should blink the on-board LED and print a message. Then, it goes to sleep.

```
rst:0x5 (DEEPSLEEP_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:2
load:0x3fff0018,len:4
load:0x3fff001c,len:4732
load:0x40078000,len:7496
load:0x40080400,len:5512
entry 0x4008114c


[0;32mI (399) cpu_start: Pro cpu up.[0m
[0;32mI (399) cpu_start: Single core mode[0m
[0;32mI (399) heap_init: Initializing. RAM available for dynamic allocation:[0m
[0;32mI (403) heap_init: At 3FFAE6E0 len 00001920 (6 KiB): DRAM[0m
[0;32mI (409) heap_init: At 3FFC0E00 len 0001F200 (124 KiB): DRAM[0m
[0;32mI (415) heap_init: At 3FFE0440 len 00003BC0 (14 KiB): D/IRAM[0m
[0;32mI (421) heap_init: At 3FFE4350 len 0001BCB0 (111 KiB): D/IRAM[0m
[0;32mI (428) heap_init: At 400915E0 len 0000EA20 (58 KiB): IRAM[0m
[0;32mI (434) cpu_start: Pro cpu start user code[0m
[0;32mI (5) cpu_start: Starting scheduler on PRO CPU.[0m


Im awake, but Im going to sleep
```

ESP8266 – Deep Sleep with Timer Wake Up

There are slightly different ways to wake up the ESP8266 with a timer after deep sleep. We recommend copying the following function to the beginning of your script, and then call the function in your code to put the ESP8266 in deep sleep mode.

```
def deep_sleep(msecs):
  #configure RTC.ALARM0 to be able to wake the device
  rtc = machine.RTC()
  rtc.irq(trigger=rtc.ALARM0, wake=machine.DEEPSLEEP)

  # set RTC.ALARM0 to fire after Xmilliseconds, waking the device
  rtc.alarm(rtc.ALARM0, msecs)

  #put the device to sleep
  machine.deepsleep()
```

This *deep_sleep()* function creates a timer that wakes up the ESP8266 after a predetermined number of seconds. To use this function later in your code, you just need to pass as an argument the sleep time in milliseconds.

In the following code, the ESP8266 is in deep sleep mode for 10 seconds, then it wakes up, blinks an LED, and goes back to sleep.

```
import machine
from machine import Pin
from time import sleep

led = Pin (2, Pin.OUT)

def deep_sleep(msecs):
  # configure RTC.ALARM0 to be able to wake the device
  rtc = machine.RTC()
  rtc.irq(trigger=rtc.ALARM0, wake=machine.DEEPSLEEP)

  # set RTC.ALARM0 to fire after Xmilliseconds, waking the device
  rtc.alarm(rtc.ALARM0, msecs)

  # put the device to sleep
  machine.deepsleep()

#blink LED
led.value(1)
sleep(1)
led.value(0)
sleep(1)

# wait 5 seconds so that you can catch the ESP awake to establish
a serial communication later
# you should remove this sleep line in your final script
sleep(5)

print('Im awake, but Im going to sleep')
# sleep for 10 seconds (10000 milliseconds)
deep_sleep(10000)
```

SOURCE CODE

https://github.com/RuiSantosdotme/ESP-MicroPython/blob/master/code/GPIOs/Deep_sleep/deep_sleep_esp8266.py

First, import the necessary libraries:

```
import machine
from machine import Pin
from time import sleep
```

Create a `Pin` object that refers to GPIO 2 called `led`. For our board, it refers to the on-board LED.

```
led = Pin (2, Pin.OUT)
```

After that, add the `deep_sleep()` function to your code:

```
def deep_sleep(msecs) :
  # configure RTC.ALARM0 to be able to wake the device
  rtc = machine.RTC()
  rtc.irq(trigger=rtc.ALARM0, wake=machine.DEEPSLEEP)

  # set RTC.ALARM0 to fire after Xmilliseconds, waking the device
  rtc.alarm(rtc.ALARM0, msecs)

  # put the device to sleep
  machine.deepsleep()
```

Then, the following lines blink the LED.

```
led.value(1)
sleep(1)
led.value(0)
sleep(1)
```

Before going to sleep, we add a delay of 5 seconds and print a message to indicate it is going to sleep.

```
sleep(5)
print('Im awake, but Im going to sleep')
```

It's important to add that delay of 5 seconds before going to sleep when we are developing the script. When you want to upload a new code to your board, it needs to be awaken. So, if you don't have the delay, it will be difficult to catch it awake to upload code. After having the final code, you can remove that delay.

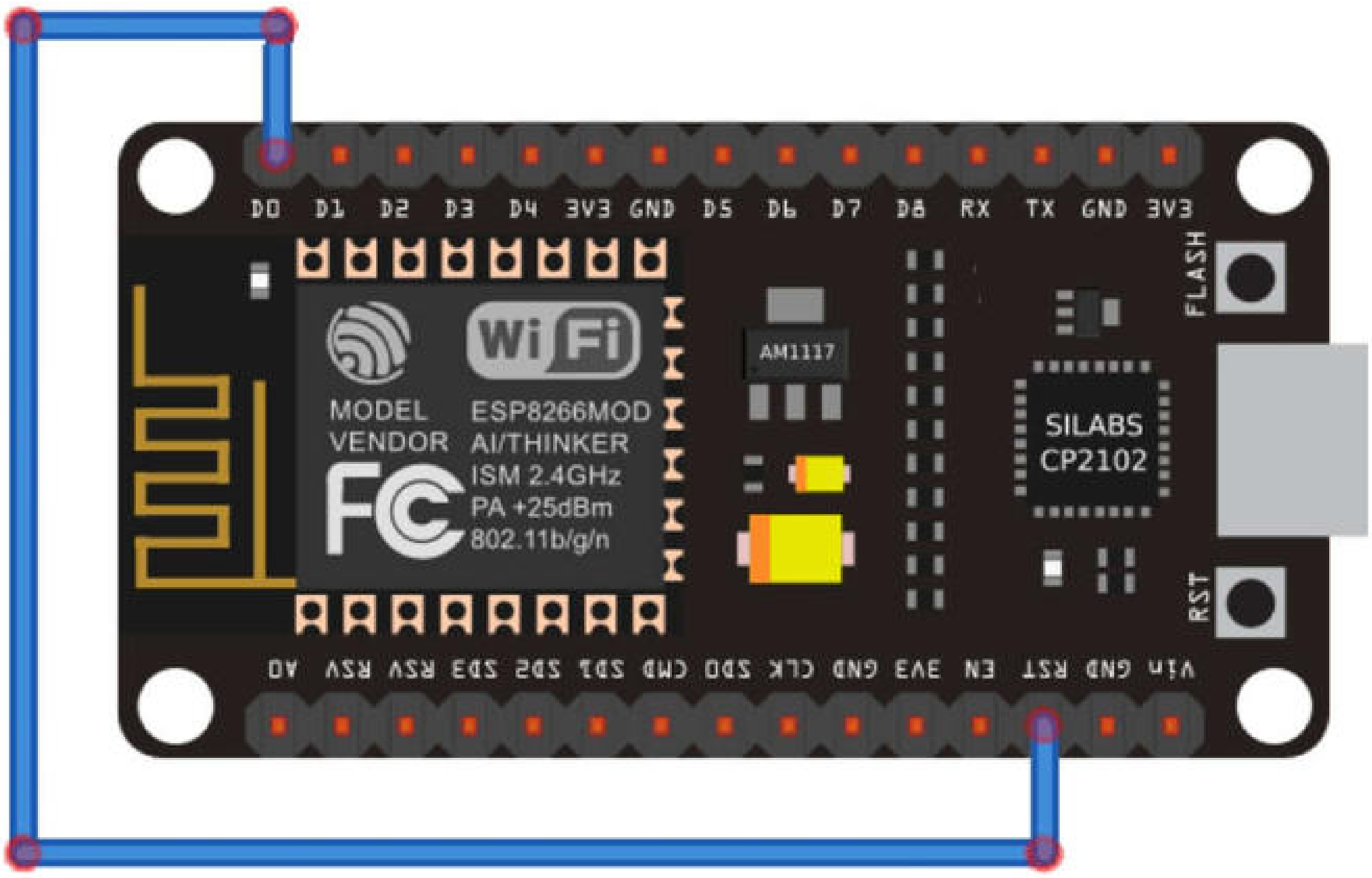
Finally, we put the ESP8266 in deep sleep for 10 seconds (10 000 milliseconds) by calling the `deep_sleep()` function and passing as argument the number of milliseconds.

```
deep_sleep(10000)
```

After 10 seconds, the ESP8266 wakes up and runs the code from the beginning, similarly of when you press the RESET button.

Demonstration

Copy the code provided to the *boot.py* file of your ESP8266 and upload the new code. After uploading the code, you need to connect GPIO 16 (D0) to the RST pin so that the ESP8266 can wake itself up.



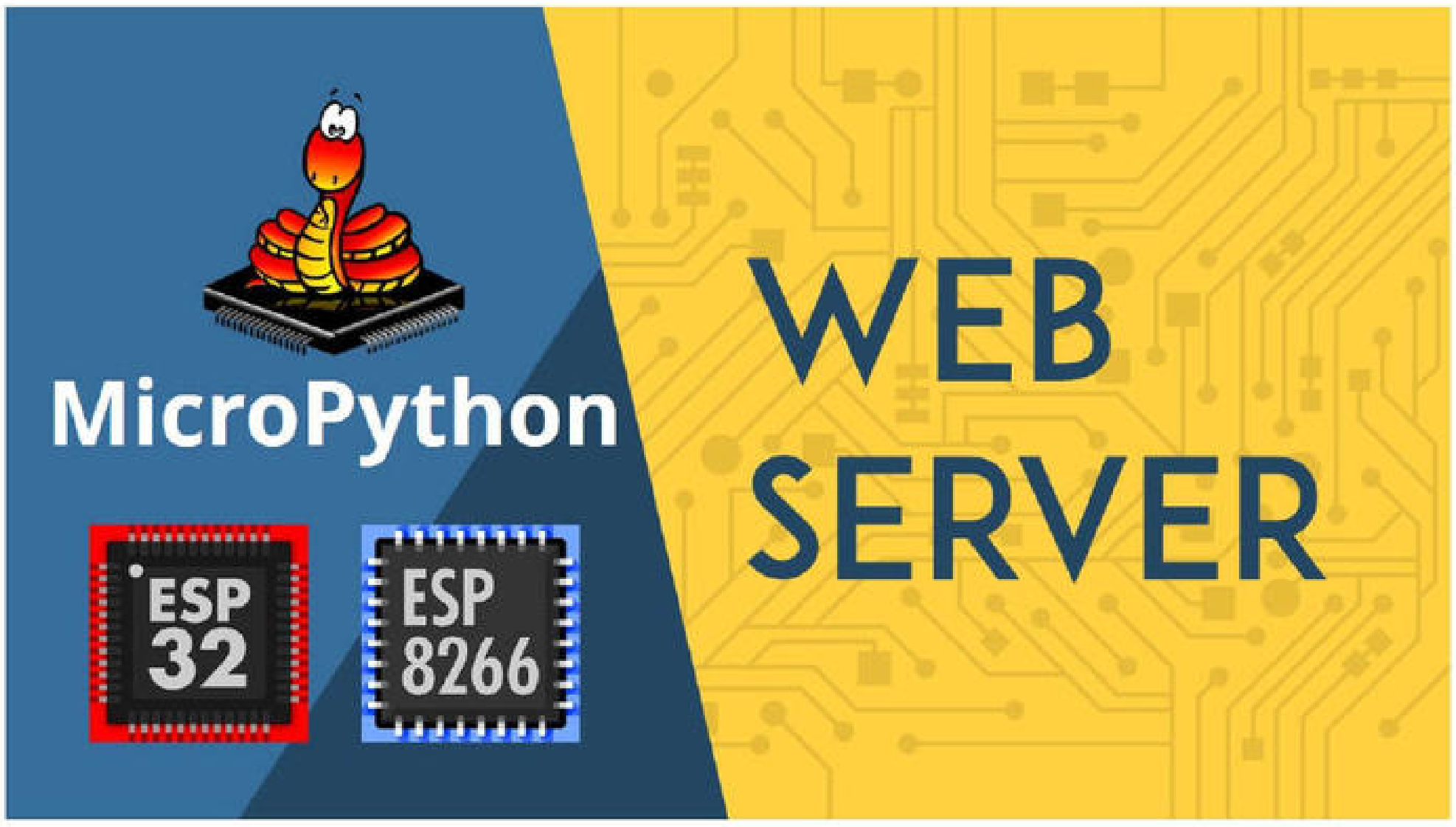
After uploading the code and connecting GPIO 16 (D0) to the RST pin, the ESP8266 should blink the on-board LED and print a message. Then, it goes to sleep for 10 seconds, wakes up and runs the code again.

```
Im awake, but Im going to sleep


$ 0l{lets_task(40100130, 3, 3fff83ec, 4)


Im awake, but Im going to sleep


d`0lslets_task(40100130, 3, 3fff83ec, 4)
```

MODULE 4

Web Servers and HTTP Clients

Web Server Introduction



In this Module we're going to make web servers and HTTP clients. We'll create a web server to remotely control outputs, a web server to display sensor readings and show you how to make HTPP requests to third-party services. We'll use IFTTT to send sensor readings to your email. Along the way we'll also show you how to deal with different types of sensors:

- DHT22/DHT11 temperature and humidity sensor
- BME280 temperature, humidity, and pressure sensor
- DS18B20 temperature sensor

Introducing Web Servers

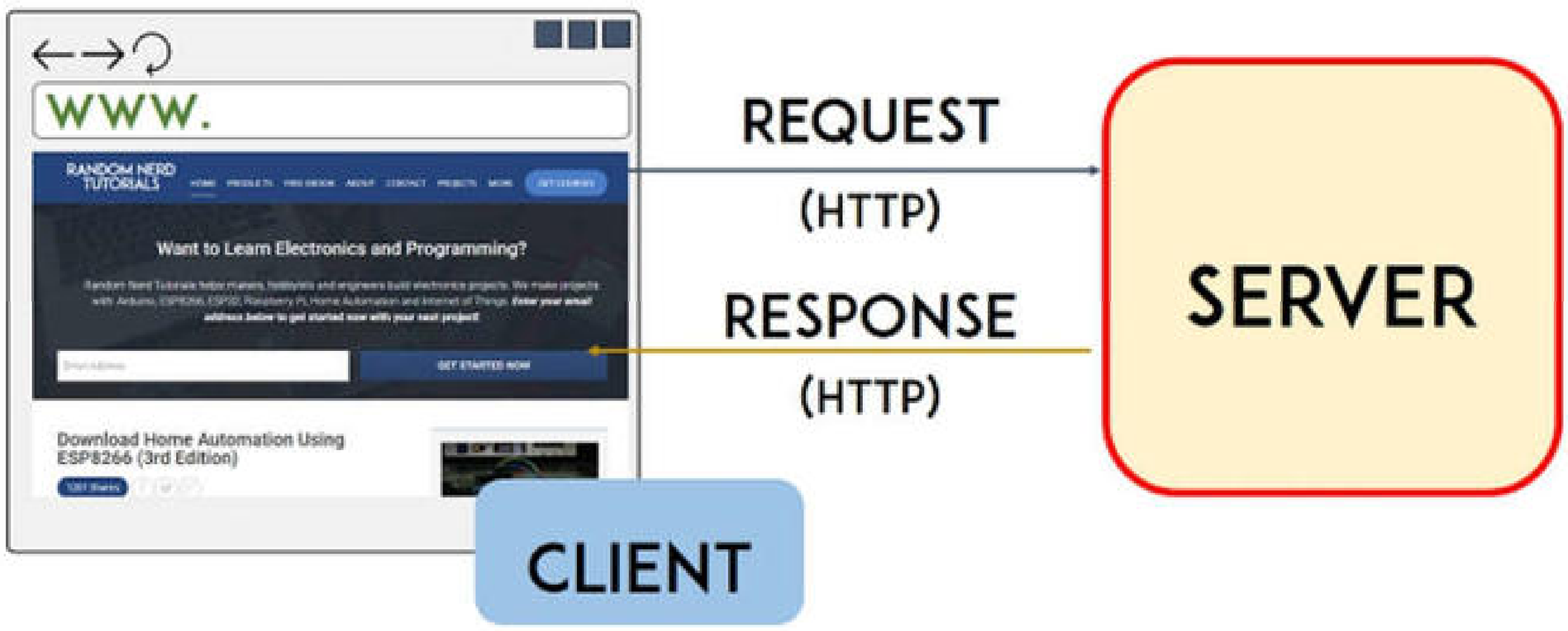
In this Unit you'll learn what a web server is and we'll look at some terms that you've probably heard before, but you may not know exactly what they mean. Let's get started!

Request-Response

Request-response is a message exchange pattern, in which a requestor sends a request message to a replier system that receives and processes the request and returns a message in response. This is a simple, yet powerful messaging pattern especially in client-server architectures.

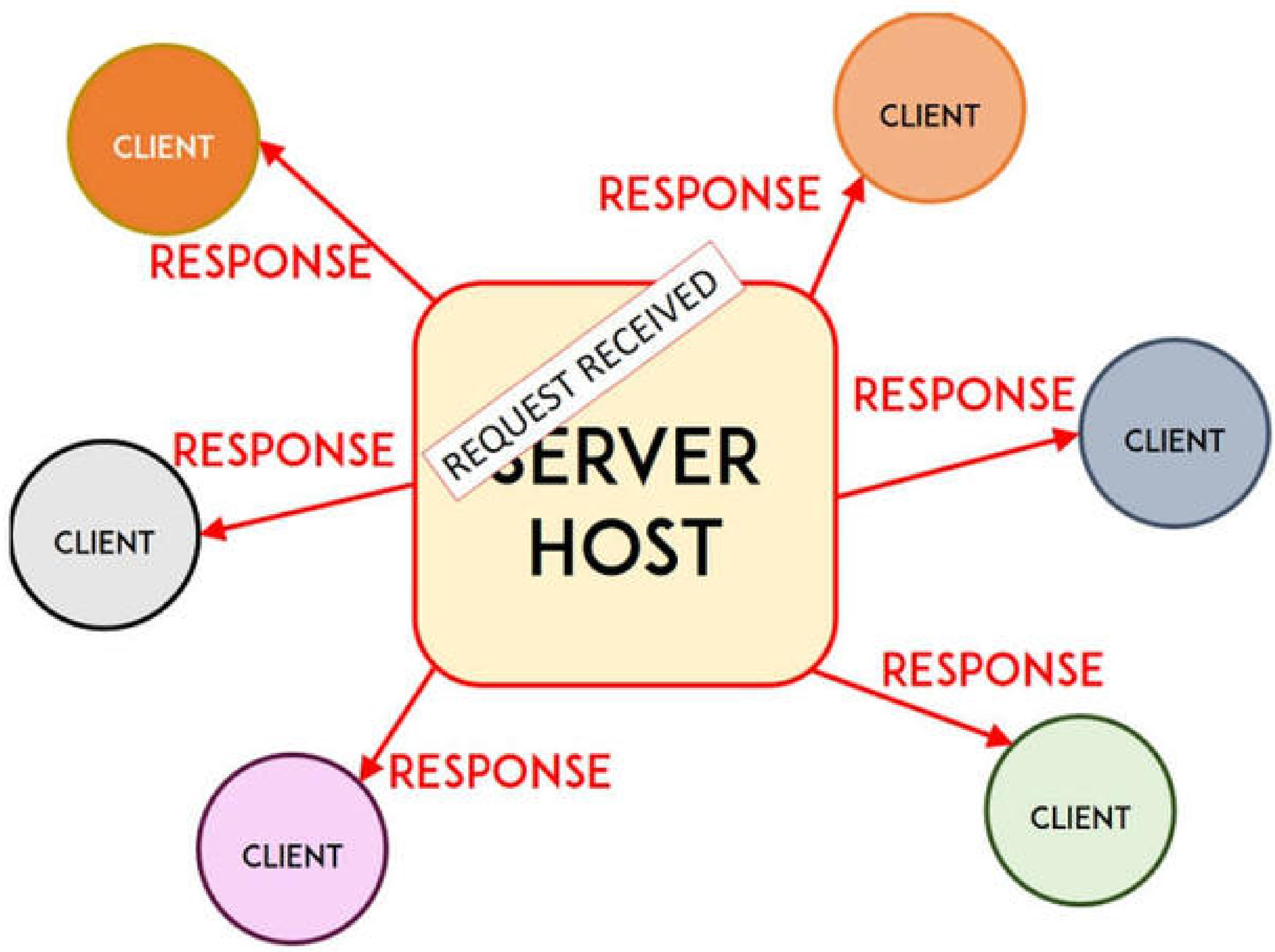
Client-Server

When you type an URL in your browser, what happens in the background is that you (the client) send a request via Hypertext Transfer Protocol (HTTP) to a server. When the server receives the request, it sends a response also through HTTP, and you see the web page you requested in your browser. Clients and servers communicate over a computer network.

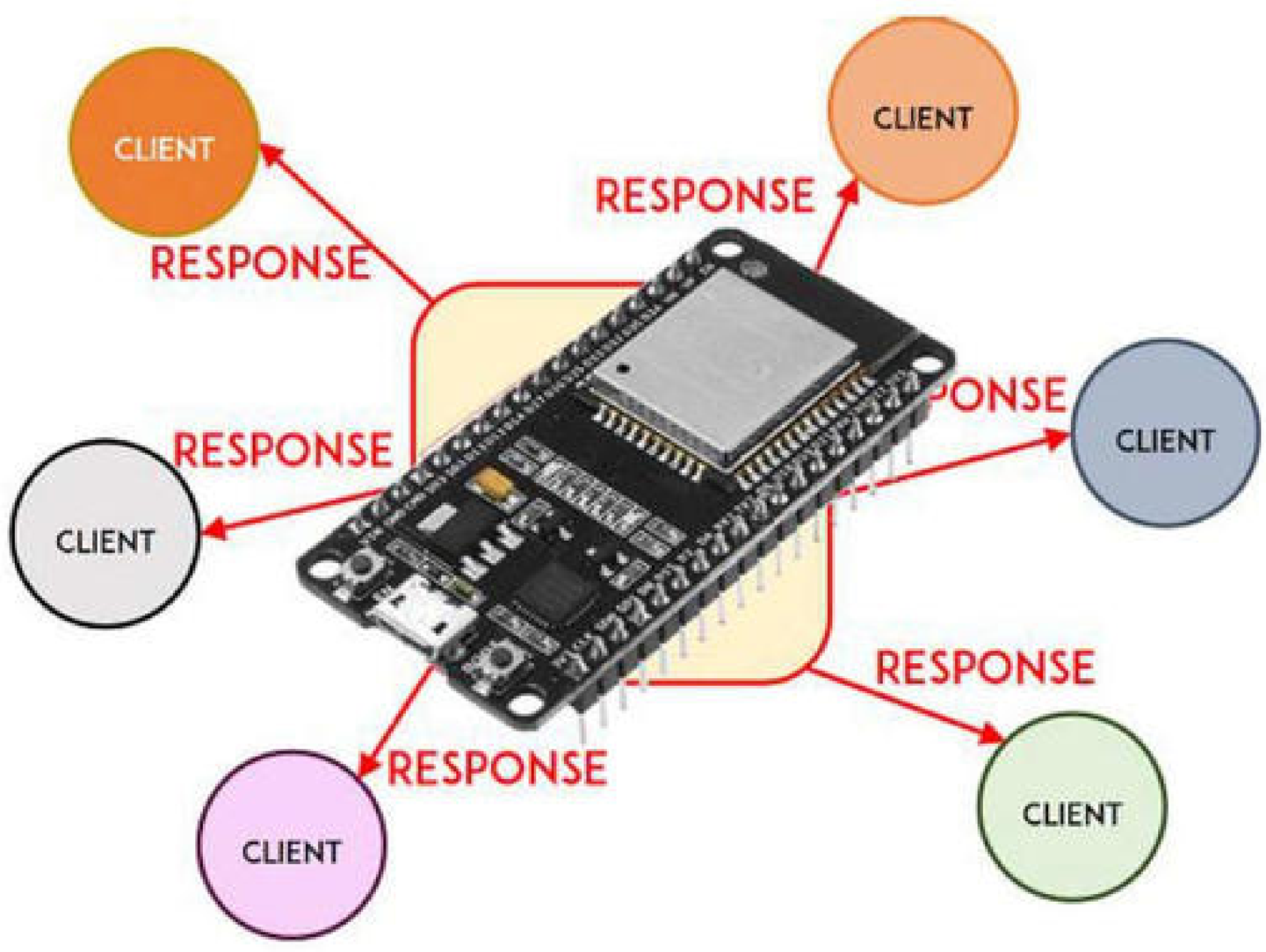


Server Host

A server host runs one or more server programs to share their resources with clients. So, you can imagine a web server as a piece of software that listens for incoming HTTP requests and sends responses when requested.

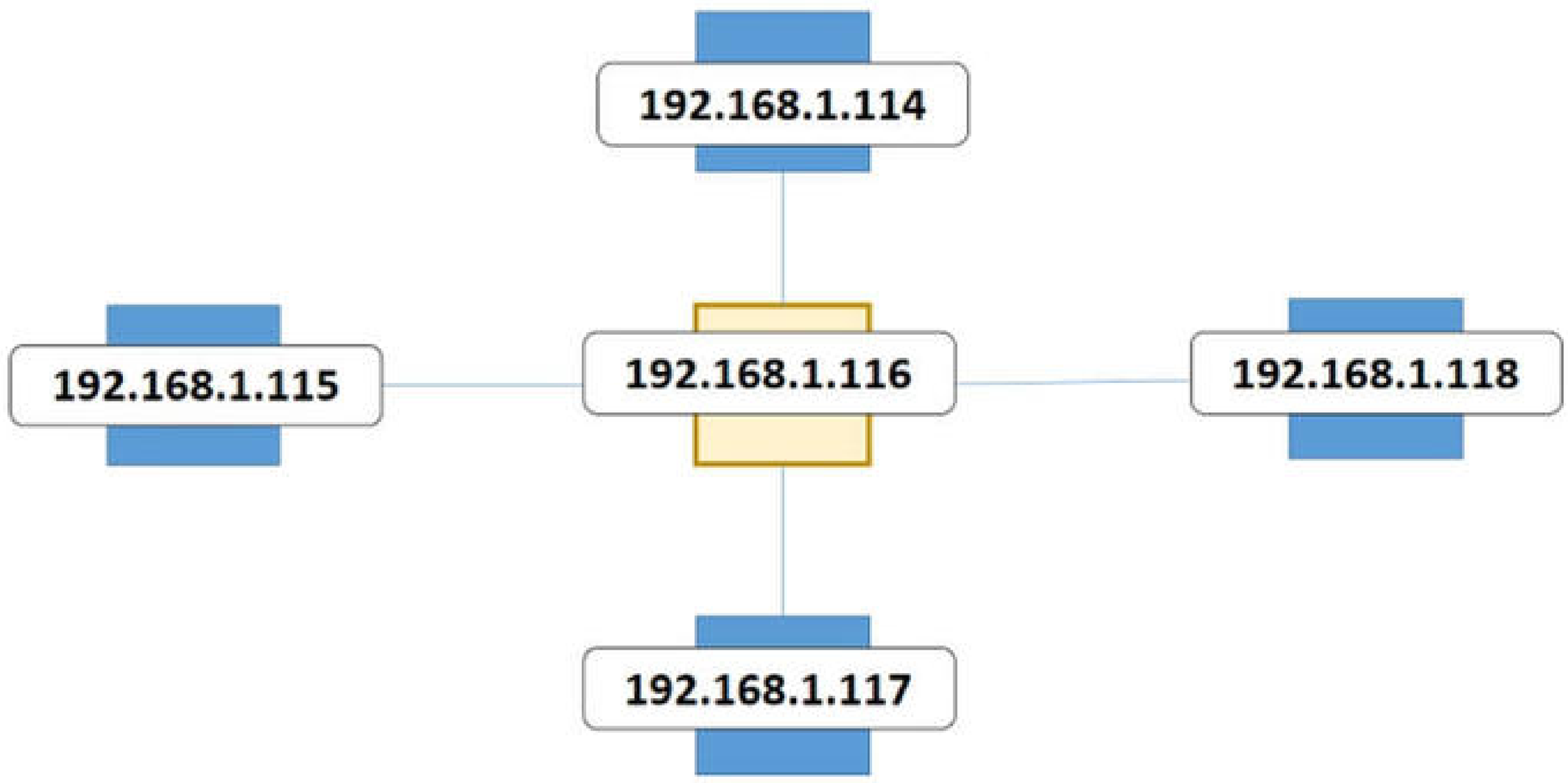


Your ESP32 or ESP8266 can act as a server host, listening for HTTP requests from clients. When a new client makes a request, the ESP sends an HTTP response.



IP Address

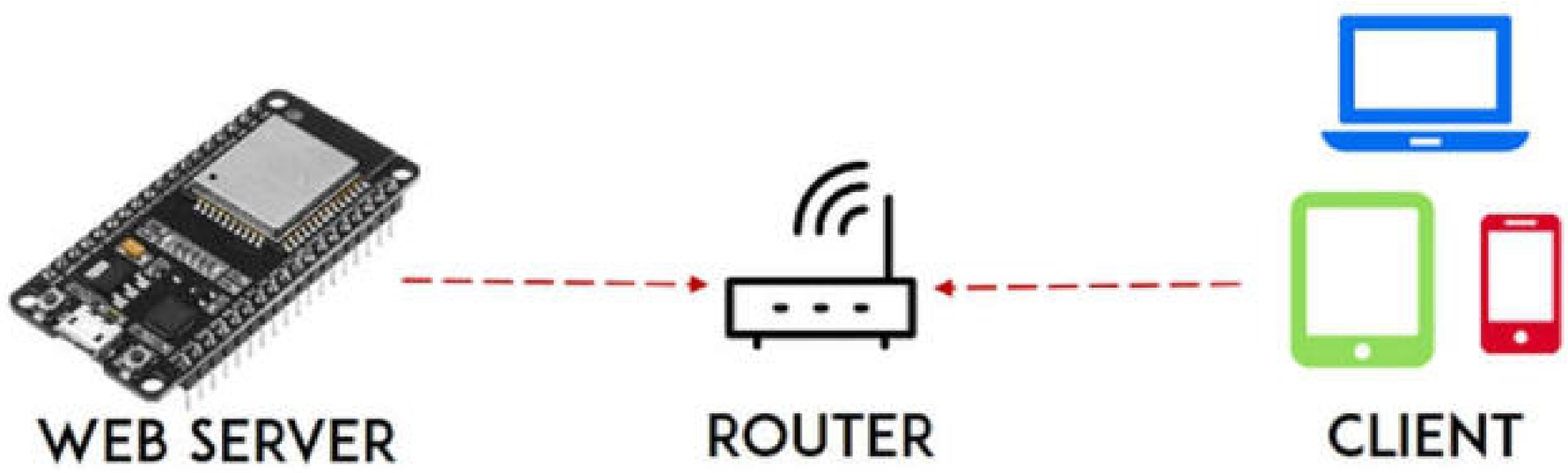
An IP address is a numerical label assigned to each device connected to a computer network. This way, any information sent to that device can reach it by referring to its IP address. So, your ESP32/ESP8266 has an IP address too.



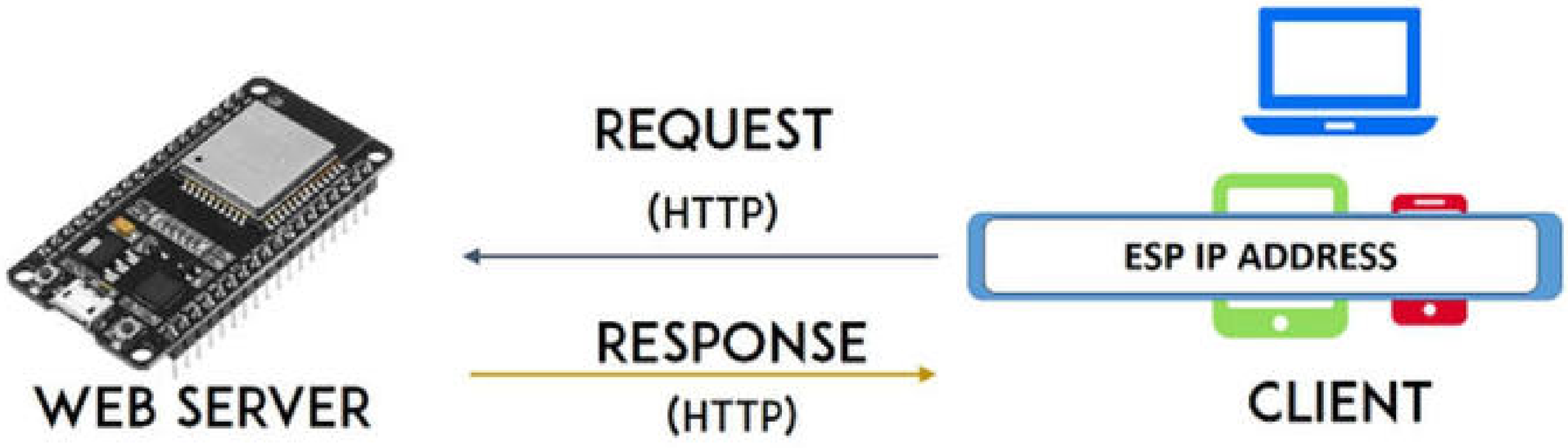
ESP32/ESP8266 Web Server

Let's look at a practical example with the ESP32/ESP8266 that can act as a web server in the local network.

Typically, a web server with the ESP32 or ESP8266 in the local network looks something like this: the ESP board running as a web server is connected via Wi-Fi to your router. Your computer, smartphone, or tablet are also connected to your router via Wi-Fi or Ethernet cable. So, the ESP and your browser are on the same network.



When you type the ESP IP address in your browser, you are sending an HTTP request to your ESP. Then, the ESP responds back with a response that can contain a value, a reading, HTML text to display a web page, or any data you program in your ESP.

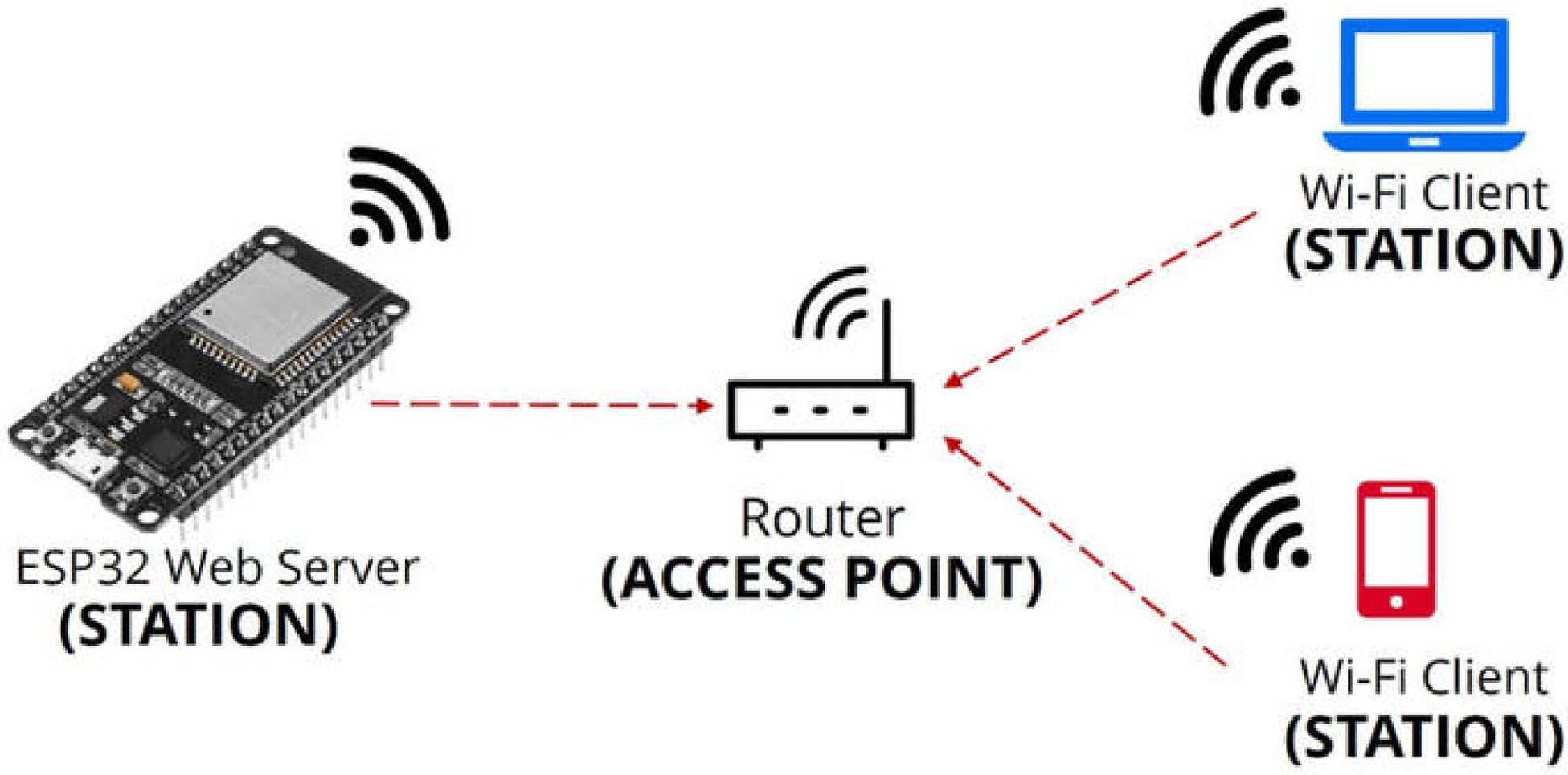


Access Point vs Station

The ESP32/ESP8266 can act as a Wi-Fi station, as an access point, or both. In the previous example, the ESP is set as a Wi-Fi station. Let's look at the differences between station and access point.

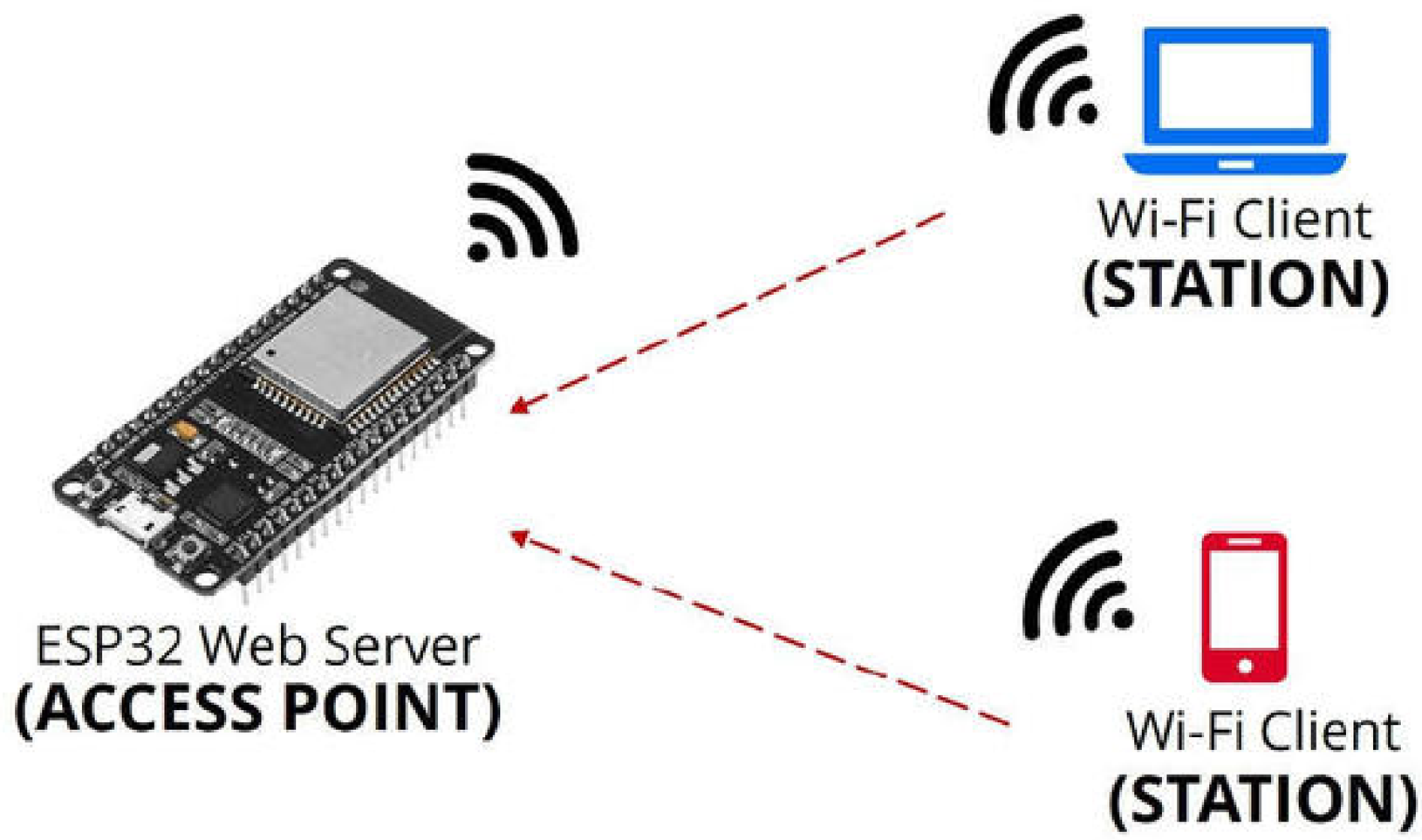
Station

When the ESP32 or ESP8266 are set as a station, we can access them through the local network. In this situation, the router acts as an access point and the ESP32 is set as a station. In this scenario, you need to be connected to your router (local network) to control the ESP.



Access Point

If you set the ESP32 or ESP8266 as an access point (hotspot), you can be connected to the ESP using any device with Wi-Fi capabilities without the need to connect to your router. In simple words, when you set the ESP as an access point you create its own Wi-Fi network and nearby Wi-Fi devices (stations) can connect to it (like your smartphone or your computer). This way, you don't need to be connected to a router to control your ESP32/ESP8266.



Because the ESP32/ESP8266 doesn't connect further to a wired network (like your router), it is called soft-AP (soft Access Point).

Sockets

We'll create our web server projects using sockets and the Python socket API.

Sockets and the socket API are used to send messages across a network. They provide a form of inter-process communication (IPC).

Sockets can be used in client-server applications, where one side acts as the server and waits for connections from clients – that's what we'll do here (the ESP32 or ESP8266 is a server waiting for connections from clients).

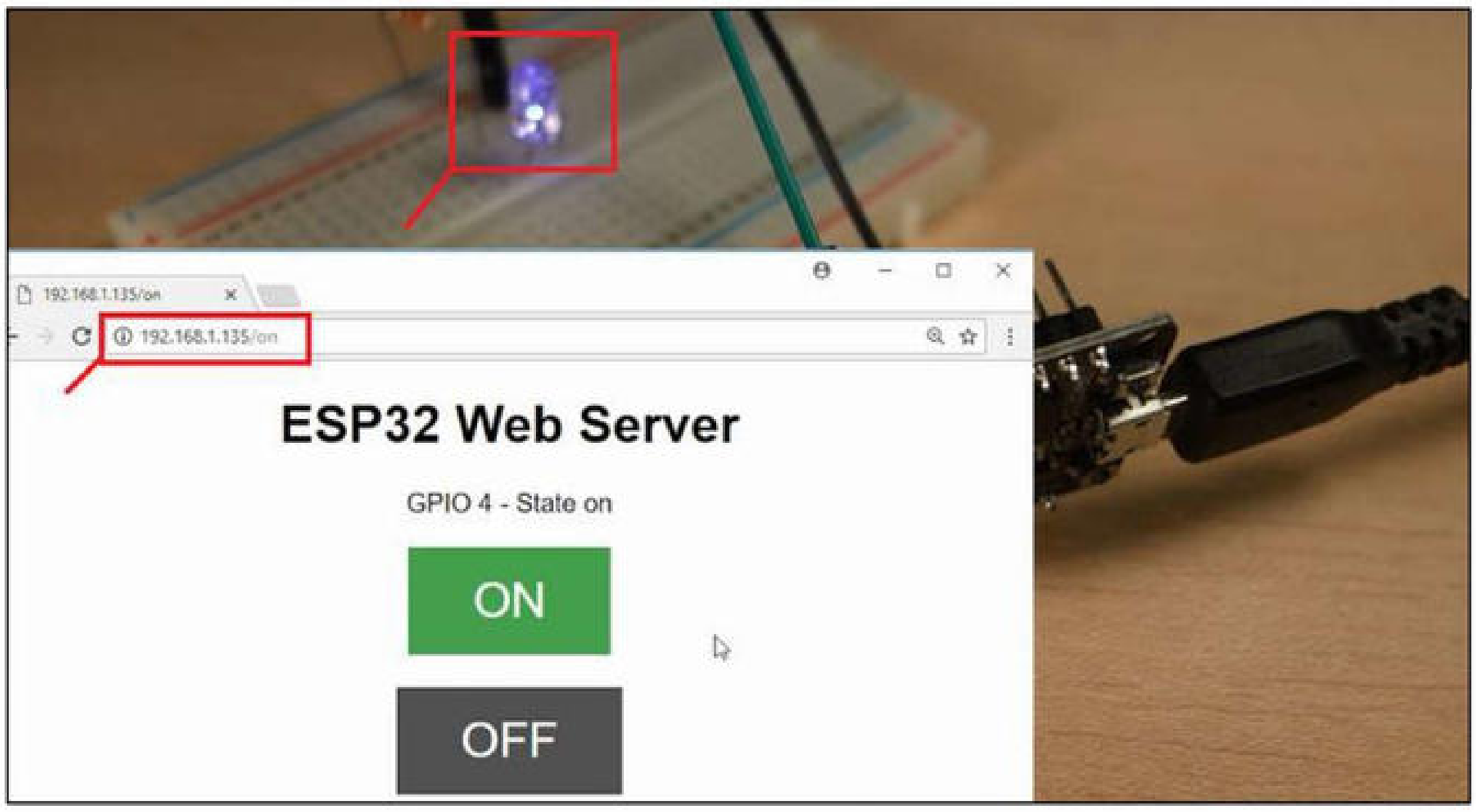
In simple words, sockets provide a way to talk to other computers (any device in your local network talking to your ESP32 or ESP8266).

Web Server Example

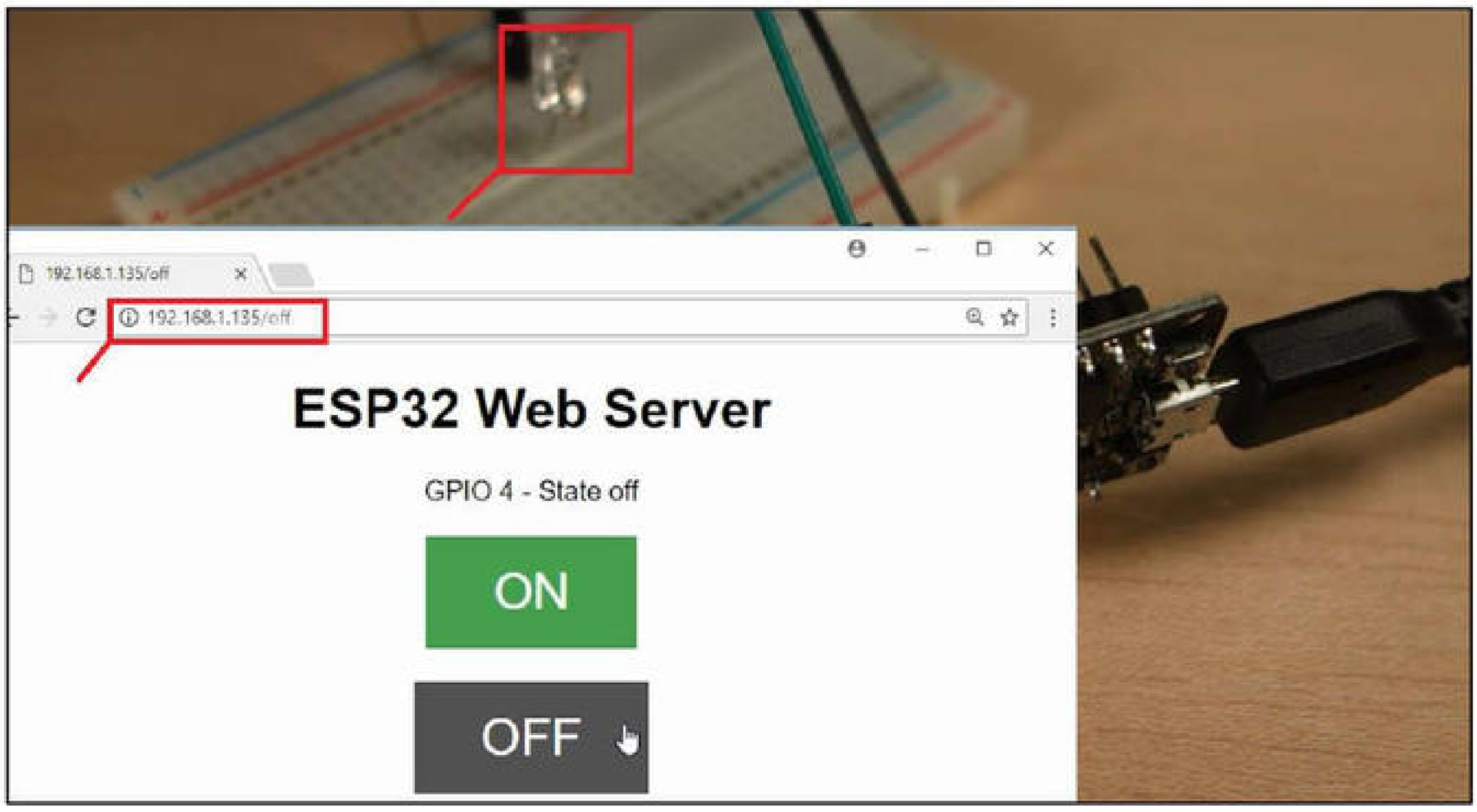
How can you put all of this together to make IoT projects with your ESP32/ESP8266? ESP32 and ESP8266 boards have GPIOs, so you can connect devices, and control them through the web.

Here's an example of a web server we built to control an output. The following web page shows up when you enter the ESP IP address in a browser.

When you press the ON button, the URL changes to the ESP IP address followed by `/on`. The ESP receives a request on that URL, so it checks with an `if` statement which URL is being requested and changes the LED state accordingly.



When you press the OFF button, a new request is made to the ESP32 in the `/off` URL. The ESP checks once again which URL is being requested and turns the LED off.



The same concept can be applied to control as many outputs as your ESP32 or ESP8266 allows you to.

Wrapping Up

This was just an overview of some important concepts to better understand ESP32/ESP8266 web servers later. The next Unit shows you how to build your very first web server.

“Hello, World!” Web Server

In this Unit, you’ll get started with web servers by building a simple “Hello, World!” web server. We’ll set the ESP32 or ESP8266 as a Wi-Fi station that is connected to your router. Then, you can access the web server by typing the ESP IP address on a browser in your local network. When you access the ESP IP address, you’ll see the “Hello, World!” web page.

Files

In our web server examples, we’ll need to write some code on the *boot.py* and *main.py* files. The *boot.py* file will contain information that only needs to run once on boot. This includes importing libraries, network credentials, instantiating pins, connecting to your network, and other configurations. The *main.py* file will contain the code that runs the web server to serve files and perform tasks based on the requests received from the client.

boot.py

With your ESP32 or ESP8266 connected to your computer, and with the uPyCraft IDE opened, double-click the *boot.py* file. Then, copy the following code:

```
try:
  import usocket as socket
except:
  import socket

import network

import esp
esp.osdebug(None)

import gc
gc.collect()

ssid = 'REPLACE_WITH_YOUR_SSID'
password = 'REPLACE_WITH_YOUR_PASSWORD'

station = network.WLAN(network.STA_IF)

station.active(True)
station.connect(ssid, password)

while station.isconnected() == False:
  pass

print('Connection successful')
print(station.ifconfig())
```

SOURCE CODE
https://github.com/RuiSantosdotme/ESP-MicroPython/blob/master/code/WiFi/Web_Server_Hello_World/boot.py

As mentioned previously, we'll create our web server using sockets and the Python *socket* API. The official documentation imports the `socket` library as follows:

```
try:
  import usocket as socket
except:
  import socket
```

Network

After importing the `socket` library, we need to import the `network` library. The `network` library allows us to connect the ESP32 or ESP8266 to a Wi-Fi network.

```
import network
```

The following lines turn off vendor OS debugging messages:

```
import esp
esp.osdebug(None)
```

Then, run a garbage collector:

```
import gc
gc.collect()
```

Note: garbage collector is a form of automatic memory management. This is a way to reclaim memory occupied by objects that are no longer in use by the program. This is useful to save space on the flash memory.

The following variables hold your network credentials:

```
ssid = 'REPLACE_WITH_YOUR_SSID'
password = 'REPLACE_WITH_YOUR_PASSWORD'
```

You should replace the words highlighted in red with your network SSID and password, so that the ESP is able to connect to your router.

Then, set the ESP32 or ESP8266 as a Wi-Fi station (we create a station interface called `station`).

```
station = network.WLAN(network.STA_IF)
```

After that, activate the `station`:

```
station.active(True)
```

Finally, the ESP32/ESP8266 connects to your router using the SSID and password defined earlier:

```
station.connect(ssid, password)
```

The following statement ensures that the code doesn't proceed while the ESP is not connected to your network.

```
while station.isconnected() == False:
  pass
```

After a successful connection, print network interface parameters like the ESP32/ESP8266 IP address – use the `ifconfig()` method on the `station` object.

```
print('Connection successful')
print(station.ifconfig())
```

These are the instructions you need to include in the *boot.py* file to connect the ESP32 or ESP8266 to your network. Whenever you need to create a web server project, you can use this *boot.py* file as a starting point.

main.py

Now, open the *main.py* file and copy the following code:

```
def web_page():
  html = """<html><head><meta name="viewport"
content="width=device-width, initial-scale=1"></head>
  <body><h1>Hello, World!</h1></body></html>"""
  return html

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind(('', 80))
s.listen(5)

while True:
  conn, addr = s.accept()
  print('Got a connection from %s' % str(addr))
  request = conn.recv(1024)
  print('Content = %s' % str(request))
  response = web_page()
  conn.send('HTTP/1.1 200 OK\n')
  conn.send('Content-Type: text/html\n')
  conn.send('Connection: close\n\n')
  conn.sendall(response)
  conn.close()
```

SOURCE CODE
https://github.com/RuiSantosdotme/ESP-MicroPython/blob/master/code/WiFi/Web_Server_Hello_World/main.py

How the Code Works

Our script starts by creating a function called `web_page()`. This function simply returns the HTML text with the necessary tags to build a "Hello, World!" web page.

```
def web_page():
  html = """<html><head><meta name="viewport"
content="width=device-width, initial-scale=1"></head>
  <body><h1>Hello, World!</h1></body></html>"""
  return html
```

Instead of creating a function, we could have simply created the `html` variable. However, in future web server projects with dynamic data, it's useful to have a function that returns a different web page depending on different conditions.

Three double quotes

Notice here that we use three double quotes " " " to enclose a string. Using three double quotes or three single quotes `'''` is used to create strings that span multiple lines. With triple quotes you can print strings on multiple lines to make lengthy text easier to read.

Understanding the HTML

If we put the HTML text in a readable format, we get the following:

```
<html><head>
  <meta name="viewport" content="width=device-width, initial-
scale=1">
</head>
<body>
  <h1>Hello, World!</h1>
</body></html>
```

SOURCE FILE
https://github.com/RuiSantosdotme/ESP-MicroPython/blob/master/code/WiFi/Web_Server_Hello_World/web_page.html

HTML, short for HyperText Markup Language, is the predominant markup language used to create web pages. Web browsers were designed to read HTML tags, which tell the browser how to display content on the page.

All the content of an HTML file needs to be sandwiched between the tags `<html>` and `</html>`, the former indicating the beginning of a web page and the latter indicating the end of the page.

HTML documents have two main parts: the head and the body. The head, which goes within the tags `<head>` and `</head>`, is where you insert data about the HTML document that is not directly visible on the page but that adds functionality to the web page, like scripts, styles, and more.

In our example, the HTML contains a `<meta>` tag to make the web page responsive in any web browser. This is something that you'll like to include in all your HTML files.

```
<meta name="viewport" content="width=device-width, initial-scale=1">
```

The body, which falls within the tags `<body>` and `</body>`, includes the content of the page, such as headings, text, images, tables, and so on.

In the body of the HTML text, we create one first heading with the "Hello, World!" text:

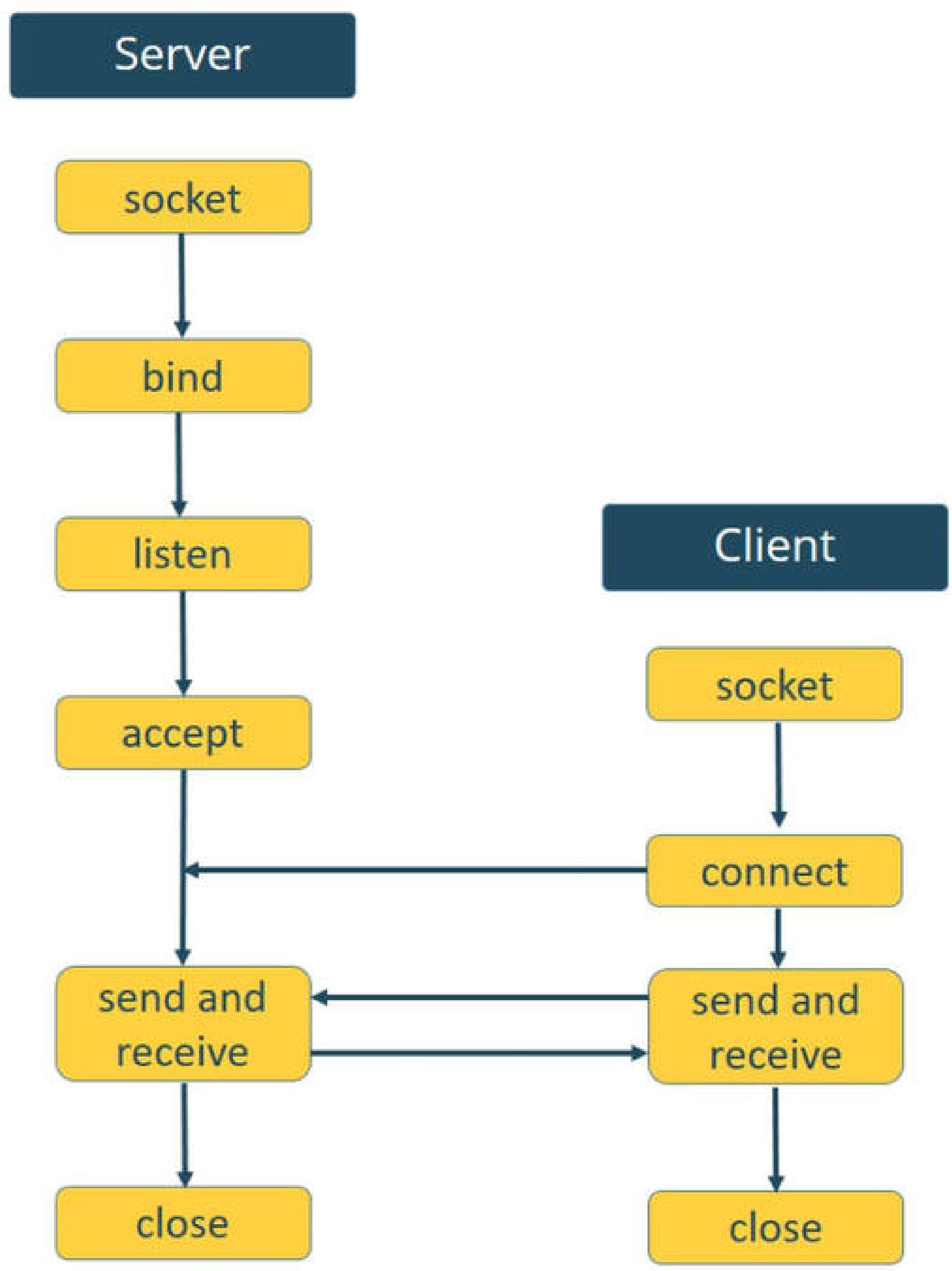
```
<h1>Hello, World!</h1>
```

Heading tags begin with an `h` followed by a number that indicates the heading level. For example, `<h1>` and `</h1>` are the tags for heading 1, or the top level; `<h2>` and `</h2>` are for heading 2, and so on until heading 6.

If you would like to add more text in a paragraph it should go between the `<p>` and `</p>` tags.

Creating a Socket

The following diagram shows how to create sockets for server-client interaction.



On the server side, first you need to create a "listening" socket. A listening socket listens for connections from clients. The socket API provides the following methods to create a "listening" socket:

- `socket()`
- `bind()`
- `listen()`
- `accept()`

In your code, the first step is to create a socket using `socket.socket()`, and specify the socket type. We create a new socket object called `s` with the given address family, and socket type. This is a STREAM TCP socket:

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

Note: Using a STREAM TCP socket is a good default to use in your projects, and you probably won't need to create other types of sockets. However, if you want to learn more on how to create other socket types, read the official documentation in the following link:

http://docs.micropython.org/en/latest/library/usocket.html

Next, we bind the socket to an address (network interface and port number) using the `bind()` method. The `bind()` method accepts a tuple variable with the IP address and port number: `bind((ip, port))`.

```
s.bind(('', 80))
```

In our example, we are passing an empty string `''` as an IP address and port 80. In this case, the empty string refers to the localhost IP address (this means the ESP32 or ESP8266 IP address).

Note: a tuple is a type of variable. It is an ordered and unchangeable collection, and it is written with round brackets. For example: `(ip, port)` is a tuple.

The next line enables the server to accept connections; it makes a "listening" socket. The argument specifies the maximum number of queued connections. The maximum is 5.

```
s.listen(5)
```

In the `while` loop is where we listen for requests and send responses. When a client connects, the server call the `accept()` method to accept the connection.

```
conn, addr = s.accept()
```

It saves a new socket object to accept and send data on the `conn` variable and saves the client address to connect to the server on the `addr` variable.

Then, print the client address and save it in the `addr` variable.

```
print('Got a connection from %s' % str(addr))
```

The data is exchanged between the client and server using the `send()` and `recv()` methods.

The following line gets the request received on the newly created socket and saves it in the `request` variable.

```
request = conn.recv(1024)
```

The `recv()` method receives the data from the client socket (remember that we've created a new socket object on the `conn` variable). The argument of the `recv()` method specifies the maximum data that can be received at once.

The next line simply prints the content of the request:

```
print('Content = %s' % str(request))
```

Then, create a variable called `response` that contains the HTML text returned by the `web_page()` function:

```
response = web_page()
```

Finally, send the response to the socket client using the `send()` and `sendall()` methods as follows:

```
conn.send('HTTP/1.1 200 OK\n')
conn.send('Content-Type: text/html\n')
conn.send('Connection: close\n\n')
conn.sendall(response)
```

In the end, close the created socket.

```
conn.close()
```

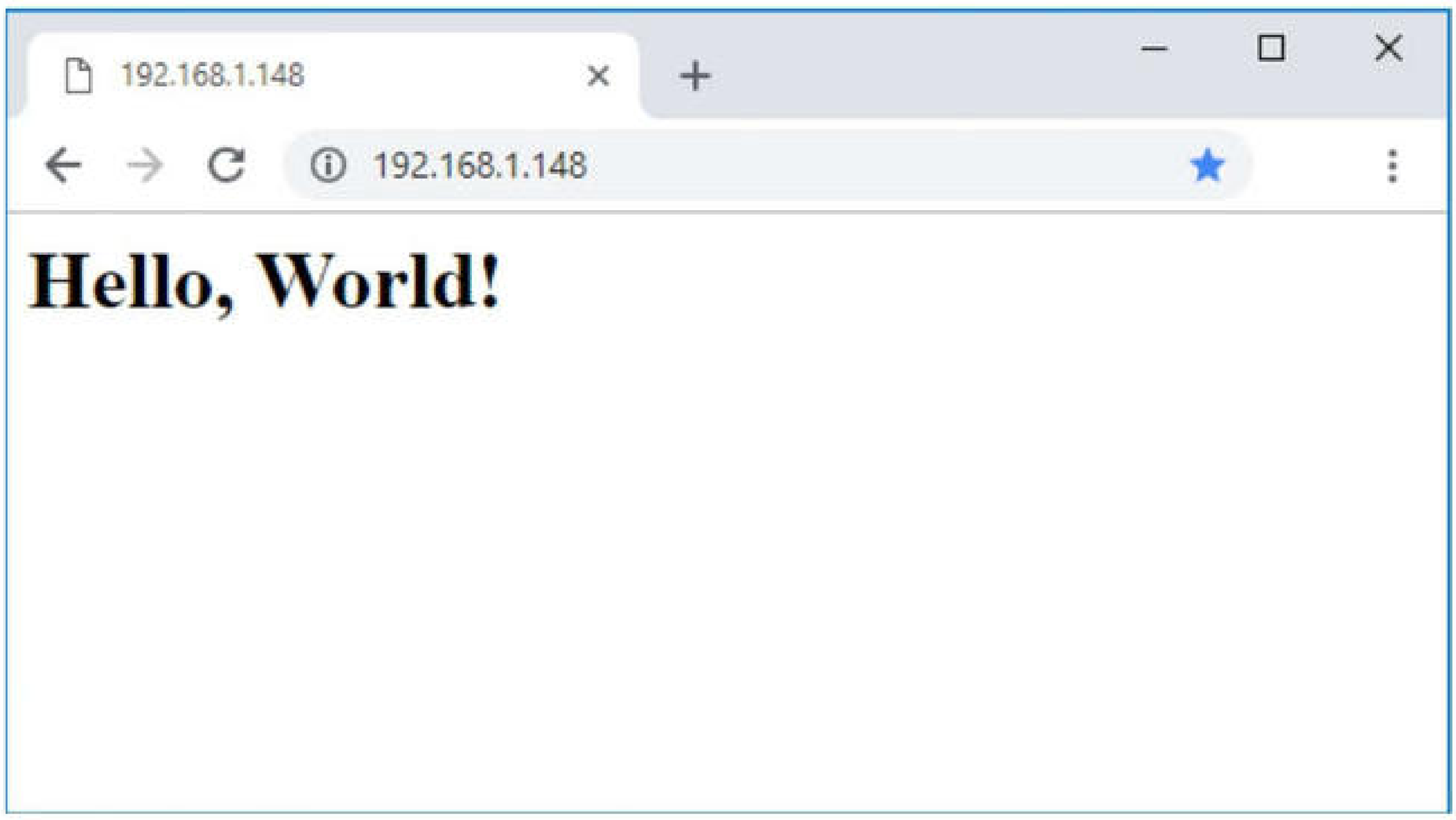
This explanation may seem a bit technical. However, you don't need to worry about what each exact line means. In our web server projects, we'll use the same methods and you'll get used to them in no time.

Demonstration

Upload the *boot.py* and *main.py* files to the ESP32 or ESP8266 board. Then, press the ESP EN/RST button. It should print the IP address on the Shell, after the "Connection successful" message, as shown in the figure below.

```
Connection successful
('192.168.1.148', '255.255.255.0', '192.168.1.254', '192.168.1.254')
```

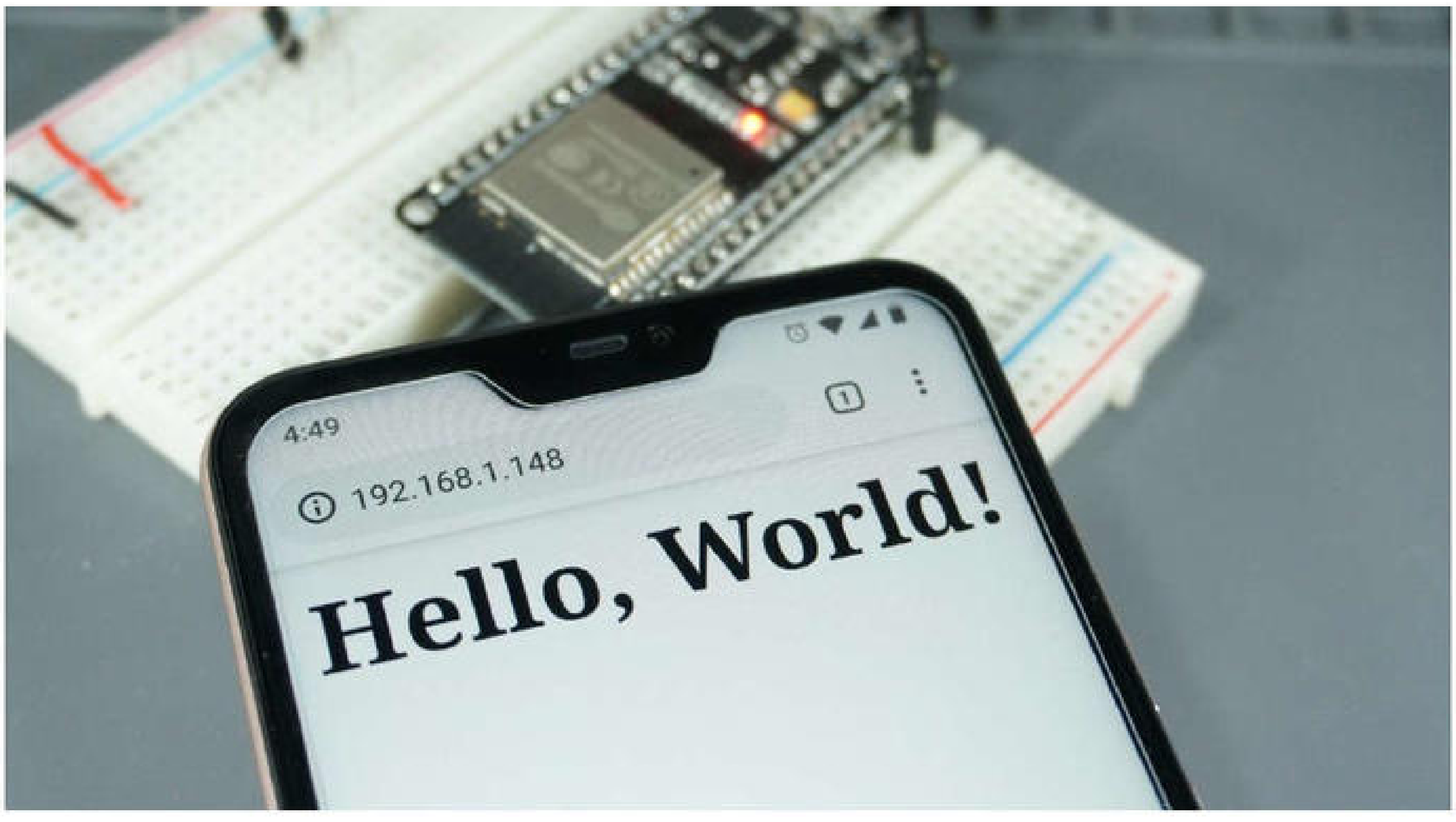
Open your browser, type the ESP IP address you've found earlier, and you should see your "Hello, World!" page.



At the same time, you should get information about the client as well as the client request (see highlights in the following figure).

```
Connection successful
('192.168.1.148', '255.255.255.0', '192.168.1.254', '192.168.1.254')
Got a connection from ('192.168.1.66', 55117)
Content = b'GET / HTTP/1.1\r\nHost: 192.168.1.148\r\nConnection: keep-alive\r\nUpgrade-Insecure-Requests: 1\r\nUser-Agent:
Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/69.0.3497.100 Safari/537.36\r\nAccept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8\r\nAccept-Encoding: gzip, deflate\r\nAccept-
Language: pt-PT,pt;q=0.9,en-US;q=0.8,en;q=0.7\r\n\r\n'
Got a connection from ('192.168.1.66', 55118)
Content = b'GET /favicon.ico HTTP/1.1\r\nHost: 192.1697.100 Safari/537.36\r\nAccept:
image/webp,image/apng,image/*,*/*;q=0.8\r\nReferer: http://192.168.1.148/\r\nAccept-Encoding: gzip, deflate\r\nAccept-Language:
pt-PT,pt;q=0.9,en-US;q=0.8,en;q=0.7\r\n\r\n'
```

You can access your web server in any device with a browser on your local network like your smartphone.


4:49
192.168.1.148
Hello, World!

Web Server – Control Outputs

In the previous Unit, you've learned the steps to create a web server to display static HTML text – the "Hello, World!" web server. In this Unit you'll learn how to create a web server that controls ESP32 or ESP8266 outputs. The web server you'll build can be accessed with any device that as a browser: smartphone, tablet, laptop, on the local network.

Project Overview

Before going straight to the project, it is important to outline what our web server will do, so that it is easier to follow the steps later in this Unit.

- The web server you'll build controls the ESP32 and ESP8266 on-board LEDs (you can control any other GPIO);
- You can access the ESP32 web server by typing the ESP32 IP address on a browser in the local network;
- By pressing the ON or OFF buttons on your web server you can instantly change the state of the LED.

The following image shows the web server you'll build.

This is one of the simplest web servers you can build to control an output: one heading, and two buttons (without any styles applied to the web page).

boot.py

Copy the following code to your *boot.py* file.

```
try:
  import usocket as socket
except:
  import socket

from machine import Pin
import network

import esp
esp.osdebug(None)

import gc
gc.collect()

ssid = 'REPLACE_WITH_YOUR_SSID'
password = 'REPLACE_WITH_YOUR_PASSWORD'

station = network.WLAN(network.STA_IF)

station.active(True)
```

```
station.connect(ssid, password)

while station.isconnected() == False:
  pass

print('Connection successful')
print(station.ifconfig())

led = Pin(2, Pin.OUT)
```

SOURCE CODE
https://github.com/RuiSantosdotme/ESP-MicroPython/blob/master/code/WiFi/Web_Server_Output_Simple/boot.py

The *boot.py* file for this project is very similar with the example in the previous Unit. The only difference is that here we need to import the `machine` module and create a `Pin` object called `led` on GPIO 2 (the ESP32 and ESP8266 on-board LED).

```
led = Pin(2, Pin.OUT)
```

Don't forget to include your network credentials in the following variables:

```
ssid = 'REPLACE_WITH_YOUR_SSID'
password = 'REPLACE_WITH_YOUR_PASSWORD'
```

main.py

Copy the following code to your *main.py* file.

```
def web_page():
  html = """<html><head><meta name="viewport"
content="width=device-width, initial-scale=1"></head>
  <body><h1>ESP Web Server</h1><a
href=\"?led=on\"><button>ON</button></a> 
  <a
href=\"?led=off\"><button>OFF</button></a></body></html>"""
  return html

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind(('', 80))
s.listen(5)

while True:
  conn, addr = s.accept()
  print('Got a connection from %s' % str(addr))
```

```
  request = conn.recv(1024)
  request = str(request)
  print('Content = %s' % request)
  led_on = request.find('/?led=on')
  led_off = request.find('/?led=off')
  if led_on == 6:
    print('LED ON')
    led.value(1)
  if led_off == 6:
    print('LED OFF')
    led.value(0)
  response = web_page()
  conn.send('HTTP/1.1 200 OK\n')
  conn.send('Content-Type: text/html\n')
  conn.send('Connection: close\n\n')
  conn.sendall(response)
  conn.close()
```

SOURCE CODE
https://github.com/RuiSantosdotme/ESP-MicroPython/blob/master/code/WiFi/Web_Server_Output_Simple/main.py

Controlling the Outputs

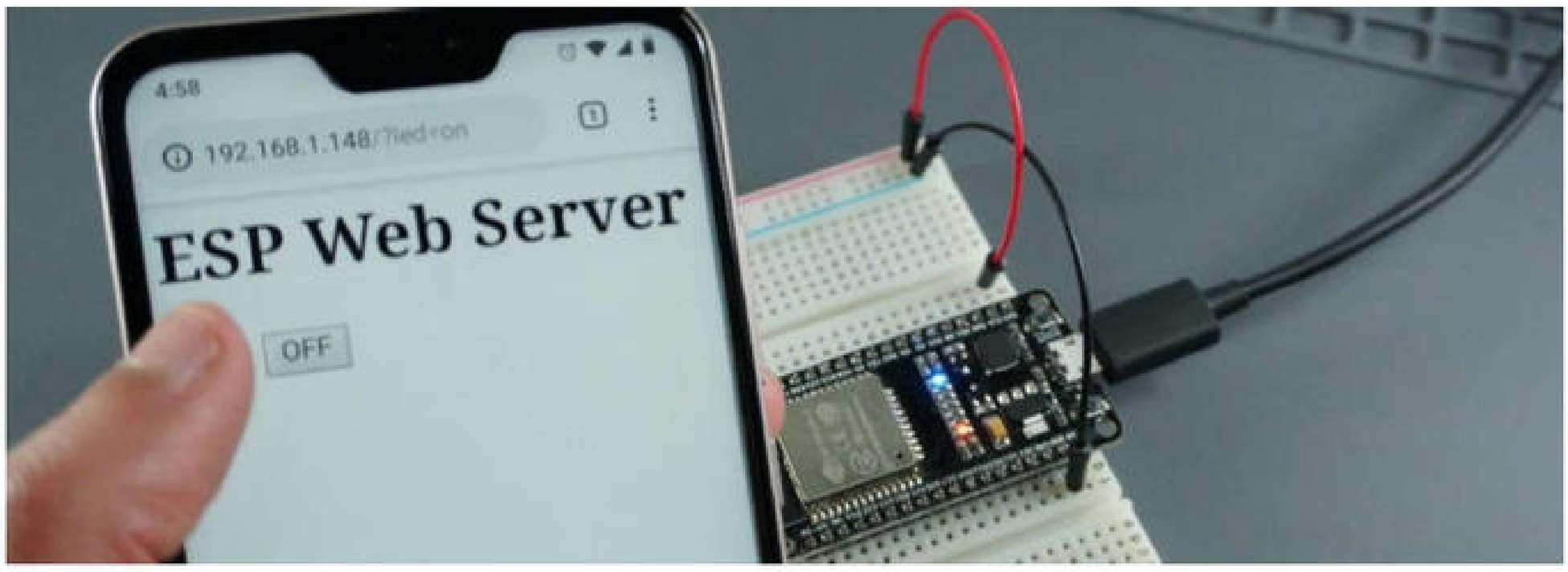
Let's see what happens when you press the web server buttons.

Upload both scripts to your ESP32 or ESP8266 board. Then, open your browser and type the ESP IP address to access the web server.

When you press the ON button you receive a request on the `/?led=on` URL.

```
Got a connection from ('192.168.1.78', 53828)
Content = b'GET /?led=on HTTP/1.1\r\nHost: 192.168.1.147\r\nConnection: keep-alive\r\nUpgrade-
Insecure-Requests: 1\r\nUser-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/70.0.3538.77 Safari/537.36\r\nAccept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8\r\nReferer
: http://192.168.1.147/\r\nAccept-Encoding: gzip, deflate\r\nAccept-Language: pt-PT,pt;q=0.9,en-
US;q=0.8,en;q=0.7\r\n\r\n'
LED ON
```

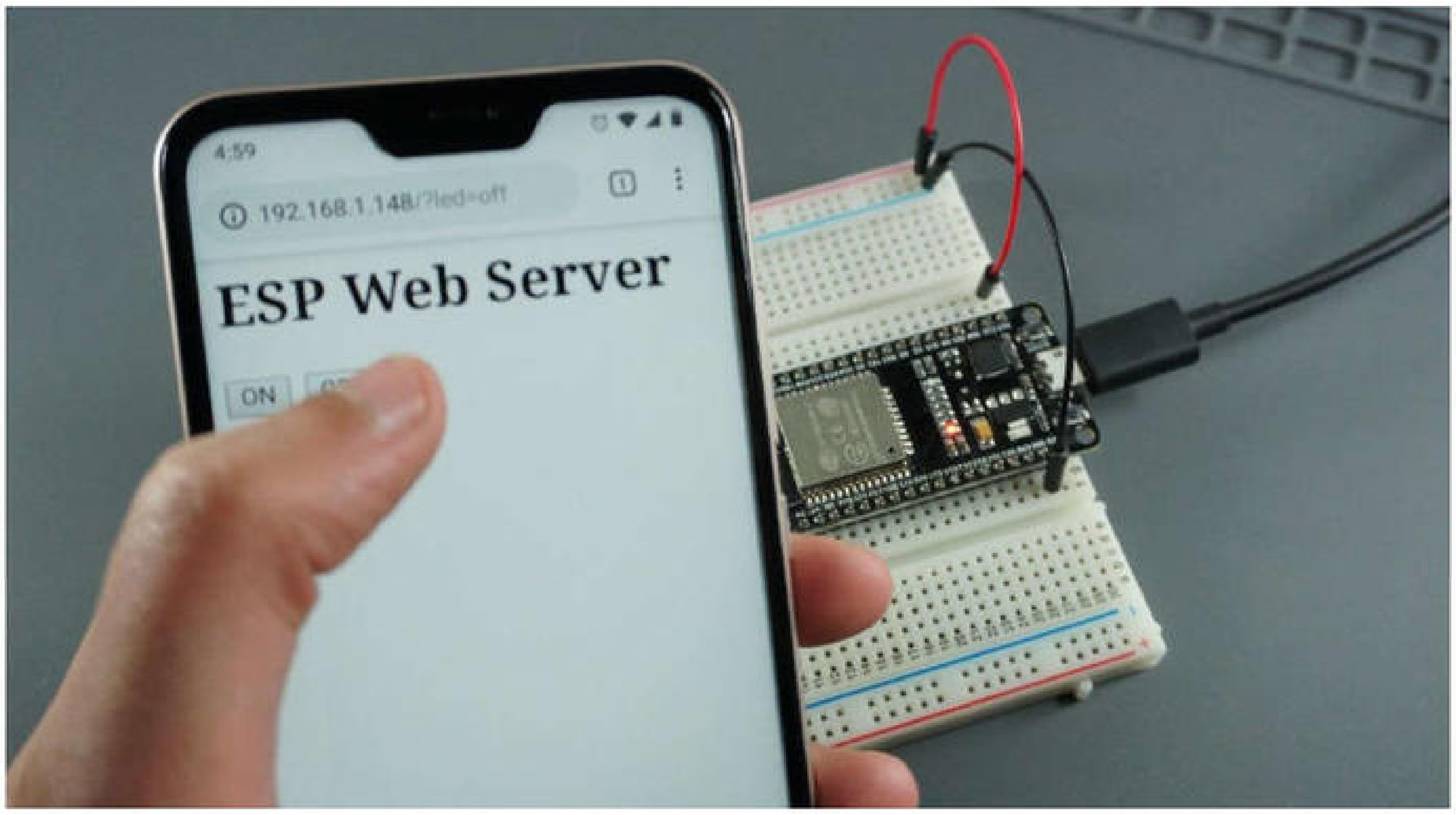
And the LED turns on.



When you press the OFF button, you receive a request on the `/?led=off` URL.

```
Got a connection from ('192.168.1.78', 53831)
Content = b'GET /?led=off HTTP/1.1\r\nHost: 192.168.1.147\r\nConnection: keep-alive\r\nUpgrade-
Insecure-Requests: 1\r\nUser-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/70.0.3538.77 Safari/537.36\r\nAccept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8\r\nReferer
: http://192.168.1.147/?led=on\r\nAccept-Encoding: gzip, deflate\r\nAccept-Language: pt-
PT,pt;q=0.9,en-US;q=0.8,en;q=0.7\r\n\r\n'
LED OFF
```

And the LED turns off.



So, depending on the received request, we can perform different tasks accordingly.

Understanding the HTML

The code starts by defining a function that returns the HTML text to build up the web page. If we rearrange the HTML text in a readable format, we get the following:

```
<html><head>
  <meta name="viewport" content="width=device-width, initial-
scale=1">
</head>
<body>
  <h1>ESP Web Server</h1>
  <a href=\"?led=on\"><button>ON</button></a> 
  <a href=\"?led=off\"><button>OFF</button></a>
</body></html>
```

SOURCE FILE
https://github.com/RuiSantosdotme/ESP-MicroPython/blob/master/code/WiFi/Web_Server_Output_Simple/web_page.html

This HTML text simply creates a page with a heading 1 and two buttons. The heading 1 contains the "ESP Web Server" text:

```
<h1>ESP Web Server</h1>
```

Then, we define two buttons: ON and OFF. When you click the ON button, you are redirected to the `/?led=on` URL:

```
<a href=\"?led=on\"><button>ON</button></a> 
```

Note: ` ` is a non-breaking space entity. A non-breaking space is a space that will not break into a new line. This way, the buttons stay together on the same line.

When you click the OFF button, you are redirected to the `/?led=off` URL:

```
<a href=\"?led=off\"><button>OFF</button></a>
```

How the Code Works

In the script, use the usual procedures to create a socket server and listening for incoming requests. To learn more about creating a socket server and listening for incoming request, read the previous Unit.

In the `while` loop, after receiving a request, we need to check if the `request` contains the `'/?led=on'` or `'/?led=off'` expressions. For that, we can apply the `find()` method on the `request` variable.

```
led_on = request.find('/?led=on')
led_off = request.find('/?led=off')
```

Note: the `find()` method returns the lowest index of the substring we are looking for. If it is not found, it returns -1.

Because the substrings we are looking for are always on index 6, we can add an if statement to detect the content of the request.

If the `led_on` variable is equal to 6, we know we've received a request on the `/?led=on` URL and we turn the LED on:

```
if led_on == 6:
  print('LED ON')
  led.value(1)
```

If the `led_off` variable is equal to 6, we've received a request on the `/?led=off` URL and we turn the LED off:

```
if led_off == 6:
  print('LED OFF')
  led.value(0)
```

Finally, send the web page to the client and close the connection:

```
response = web_page()
conn.send('HTTP/1.1 200 OK\n')
conn.send('Content-Type: text/html\n')
conn.send('Connection: close\n\n')
conn.sendall(response)
conn.close()
```

In summary, to control your outputs, you need to make different requests to your ESP32 or ESP8266 board, and act accordingly to the requested content.

Wrapping Up

Now that you know how the code works, you can modify the code to add more outputs. Also, instead of an LED, you can control a relay module, or any other electronics appliances.

You can also add CSS and JavaScript to your web page to change the way it looks and add other functionality.

In the next example, we'll build a web server with the same functionalities but with a different look (this will include CSS and JavaScript).

Web Server with Slider Switch

In this Unit we'll create a web server to control the ESP8266/ESP32 on-board LED.

Instead of the buttons used in the previous Unit, we'll use a slider button.

Project Overview

Let's take a quick look at the web server you'll build and how it looks:

- The web server contains an element of type checkbox (the slider button) that controls the LED;
- When the checkbox is checked (slider to the right), the LED is on:
- When the checkbox is not checked (slider to the left in grey color), the LED is off.

boot.py

Copy the following code to your *boot.py* file.

```
try:
  import usocket as socket
except:
  import socket

from machine import Pin
import network

import esp
esp.osdebug(None)

import gc
gc.collect()

ssid = 'REPLACE_WITH_YOUR_SSID'
password = 'REPLACE_WITH_YOUR_PASSWORD'

station = network.WLAN(network.STA_IF)

station.active(True)
station.connect(ssid, password)

while station.isconnected() == False:
  pass

print('Connection successful')
print(station.ifconfig())

led = Pin(2, Pin.OUT)
```

SOURCE CODE
https://github.com/RuiSantosdotme/ESP-MicroPython/blob/master/code/WiFi/Web_Server_Output/boot.py

The *boot.py* script is the same of the previous Unit, you can go to the previous Unit to learn how it works.

main.py

Copy the following code to your *main.py* file.

```
def web_page():
  if led.value() == 1:
    led_state = 'checked'
  else:
    led_state = ""
  html = """<html><head><meta name="viewport" content="width=device-
width, initial-scale=1"><style>
  body{max-width: 300px; margin: 0px auto;}
  .switch{position:relative;display:inline-
block;width:120px;height:68px}.switch input{display:none}
  .slider{position:absolute;top:0;left:0;right:0;bottom:0;background-
color:#ccc;border-radius:34px}

.slider:before{position:absolute;content:"";height:52px;width:52px;left:8px
;bottom:8px;background-color:#fff;-webkit-
transition:.4s;transition:.4s;border-radius:68px}
  input:checked+.slider{background-color:#2196F3}
  input:checked+.slider:before{-webkit-transform:translateX(52px);-ms-
transform:translateX(52px);transform:translateX(52px)}
  </style><script>function toggleCheckbox(element) { var xhr = new
XMLHttpRequest(); if(element.checked){ xhr.open("GET", "/?led=on", true); }
  else { xhr.open("GET", "/?led=off", true); } xhr.send();
}</script></head><body>
  <h1>ESP Web Server</h1><label class="switch"><input type="checkbox"
onchange="toggleCheckbox(this)" %s><span class="slider">
  </span></label></body></html>""" % (led_state)
  return html

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind(('', 80))
s.listen(5)

while True:
  conn, addr = s.accept()
  print('Got a connection from %s' % str(addr))
  request = conn.recv(1024)
  request = str(request)
  print('Content = %s' % request)
  led_on = request.find('/?led=on')
  led_off = request.find('/?led=off')
  if led_on == 6:
    print('LED ON')
    led.value(1)
  if led_off == 6:
    print('LED OFF')
```

```
    led.value(0)
  response = web_page()
  conn.send('HTTP/1.1 200 OK\n')
  conn.send('Content-Type: text/html\n')
  conn.send('Connection: close\n\n')
  conn.sendall(response)
  conn.close()
```

SOURCE CODE
https://github.com/RuiSantosdotme/ESP-MicroPython/blob/master/code/WiFi/Web_Server_Output/main.py

Understanding the HTML

The `html` variable contains the text to build the web page. If we put the HTML text in a readable format, we get the following:

```
<html><head>
<meta name="viewport" content="width=device-width, initial-scale=1">
  <style>
    body {
      max-width: 300px;
      margin: 0px auto;
    }
    .switch {
      position:relative;
      display:inline-block;
      width:120px;
      height:68px
    }
    .switch input { display:none }
    .slider {
      position:absolute;
      top:0;
      left:0;
      right:0;
      bottom:0;
      background-color:#ccc;
      border-radius:34px
    }
    .slider:before {
      position:absolute;
      content:"";
      height:52px;
      width:52px;
```

```
    left:8px;
    bottom:8px;
    background-color:#fff;
    -webkit-transition:.4s;
    transition:.4s;
    border-radius:68px}
  input:checked+.slider{ background-color:#2196F3 }
  input:checked+.slider:before {
    -webkit-transform:translateX(52px);
    -ms-transform:translateX(52px);
    transform:translateX(52px)
  }
</style>
<script>
  function toggleCheckbox(element) {
    var xhr = new XMLHttpRequest();
    if(element.checked){xhr.open("GET", "/?led=on", true); }
    else { xhr.open("GET", "/?led=off", true); }
    xhr.send();
  }
</script>
</head><body>
<h1>ESP Web Server</h1>
<label class="switch">
  <input type="checkbox" onchange="toggleCheckbox(this)" %s>
  <span class="slider"></span>
</label>
</body></html>
```

SOURCE FILE
https://github.com/RuiSantosdotme/ESP-MicroPython/blob/master/code/WiFi/Web_Server_Output/web_page.html

We've highlighted the HTML text with three different colors. The text highlighted in red between the `<style>` and `</style>` tags, is CSS and it's used to style the web page. Most of the CSS text in the script formats the appearance of the slider switch.

The text highlighted in blue is JavaScript, and it is between the `<script>` and `</script>` tags.

The remainder text highlighted in green, corresponds to the HTML that builds up the web page: the heading and the slider switch.

The slider switch is added to the web page with the following lines:

```
<label class="switch">
    <input type="checkbox" onchange="toggleCheckbox(this)" %s>
    <span class="slider"></span>
  </label>
```

The slider switch is an input of type "**checkbox**". Besides the `type` attribute you also pass the `onchnage` attribute. The `onchange` attribute specifies what happens when there's a change on the slider state. In this case, it calls the `toggleCheckbox` function defined in the JavaScript code.

The checkbox can have two states: checked or not checked.

For a non-checked checkbox, you have the following:

```
<input type="checkbox" onchange="toggleCheckbox(this)">
```

To create the checked checkbox, you need to add the word "checked" to the `<input>` tag as follows:

```
<input type="checkbox" onchange="toggleCheckbox(this)" checked>
```

Because this changes depending on the LED state, add a **%s** sign to be replaced by the variable value. This variable tells whether we add a "checked" expression or not. In this case, that is handled by the `led_state` variable.

Then, the JavaScript code makes an HTTP GET request on the `/?led=on` URL when the checkbox is checked, or on the `/?led=off` URL when is not checked.

```
function toggleCheckbox(element) {
  var xhr = new XMLHttpRequest();
  if(element.checked){xhr.open("GET", "/?led=on", true); }
  else { xhr.open("GET", "/?led=off", true); }
  xhr.send();
}
```

How the Code Works

Now, let's go back to the script. Because the checkbox state changes accordingly to the LED state, create a function called `web_page()` that generates different web pages depending on the LED state.

If the LED state is 1, change the `led_state` variable to `'checked'`:

```
if led.value() == 1:
  led_state = 'checked'
```

Otherwise, it should be an empty string:

```
else:
  led_state = ""
```

In the end of the `html` string, we add the `%(led_state)` that will replace the `%s` with the content of the `led_state` variable:

```
(...)
</span></label></body></html>""" % (led_state)
```

Finally, return the `html` variable:

```
return html
```

After that, we add the usual procedures to create a socket server and listening for incoming requests.

We need to check if the `request` contains the `'/?led=on'` or `'/?led=off'` expressions. For that, we can apply the `find()` method on the `request` variable.

```
led_on = request.find('/?led=on')
led_off = request.find('/?led=off')
```

Note: the `find()` method returns the lowest index of the substring we are looking for. If it's not found, it returns -1.

Because the substrings we are looking for are always on index 6, we can add an `if` statement to detect the content of the request.

If the `led_on` variable is equal to 6, we've received a request on the `/?led=on` URL and we turn the LED on:

```
if led_on == 6:
  print('LED ON')
  led.value(1)
```

If the `led_off` variable is equal to 6, we've received a request on the `/?led=off` URL and we turn the LED off:

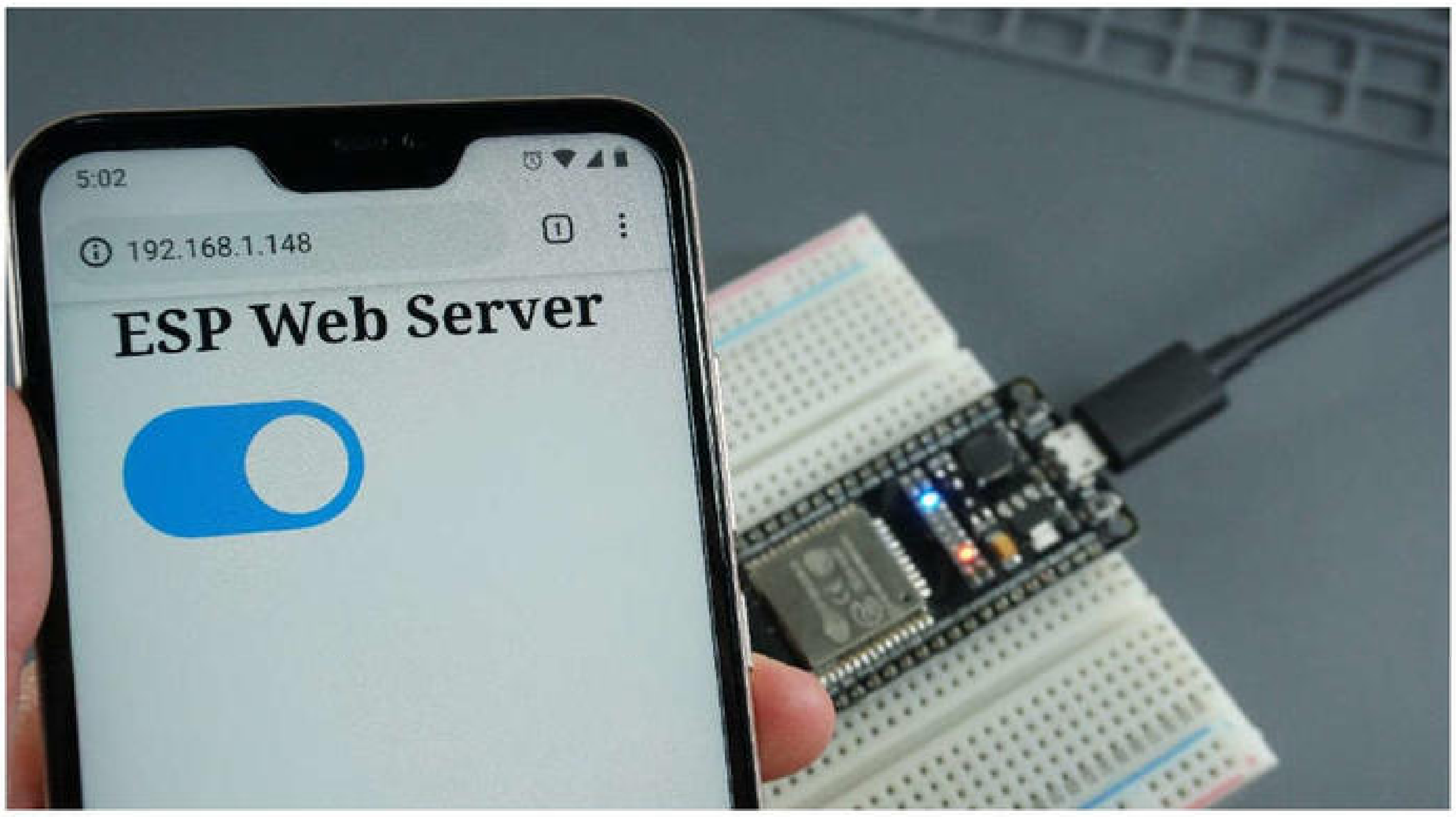
```
if led_off == 6:
  print('LED OFF')
  led.value(0)
```

Finally, send the web page to the client and close the connection:

```
response = web_page()
conn.send('HTTP/1.1 200 OK\n')
conn.send('Content-Type: text/html\n')
conn.send('Connection: close\n\n')
conn.sendall(response)
conn.close()
```

Demonstration

After uploading the b*oot.py* and *main.py* files to your ESP32 or ESP8266 board, open your browser and go to the ESP IP address.



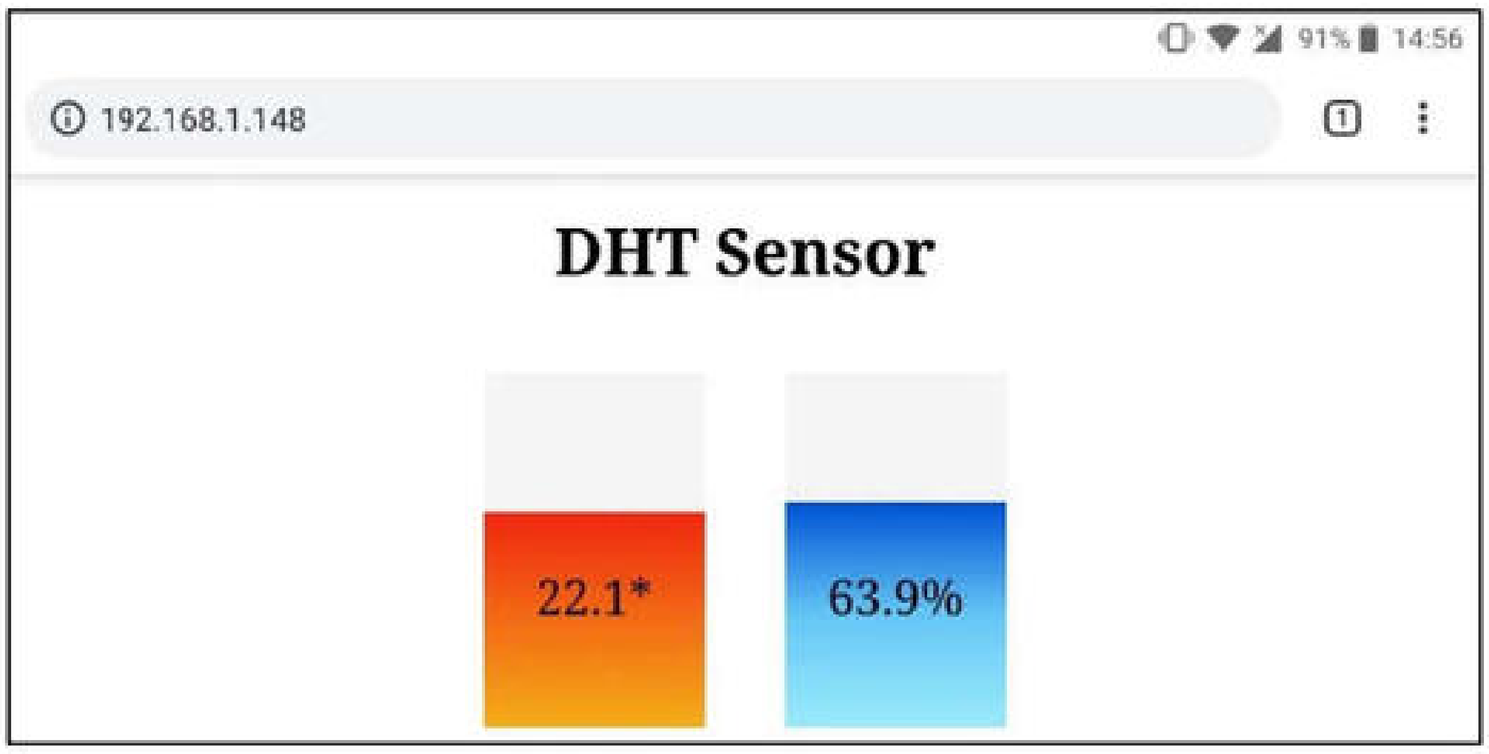
You should see a web server with a slider switch to control the on-board LED on and off. This is just another way to control an output with a nicer interface.

Web Server - Display Temperature and Humidity Readings

In this Unit, you're going to learn how to create a simple web server with the ESP32 or ESP8266 to display readings from the DHT11/DHT22 temperature and humidity sensor. Besides learning how to display data on the web server page, you'll also learn how to use the DHT11/DHT22 temperature sensor.

Project Overview

This web server displays temperature and humidity readings in progress bars as shown in the following figure:



The bars move depending on the temperature and humidity values. The exact temperature and humidity values are also displayed in the progress bar.

DHT11/DHT22 Temperature and Humidity Sensor

The DHT11 and DHT22 sensors are very popular among makers. The DHT sensors are inexpensive sensors for measuring temperature and humidity. The following figure shows the DHT22 temperature sensor.

These sensors contain a chip that does analog to digital conversion and spits out a digital signal with the temperature and humidity. The DHT11 and DHT22 are very similar, but the DHT22 is a bit more accurate. The following table shows the DHT22/DHT11 pinout. When the sensor is facing you, pin numbering starts at 1 from left to right:

DHT pin	Connect to
1	3.3V
2	Any digital GPIO; also connect a 4.7k Ohm pull-up resistor
3	Don't connect
4	GND

Schematic

For this project you need to wire the DHT11 or DHT22 temperature sensor to the ESP32 or ESP8266. You need to use a 4.7k Ohm pull-up resistor.

Note: some DHT11/DHT22 sensors are available as a module. These already come with the pull-up resistor, so you don't need to add the resistor to the circuit.

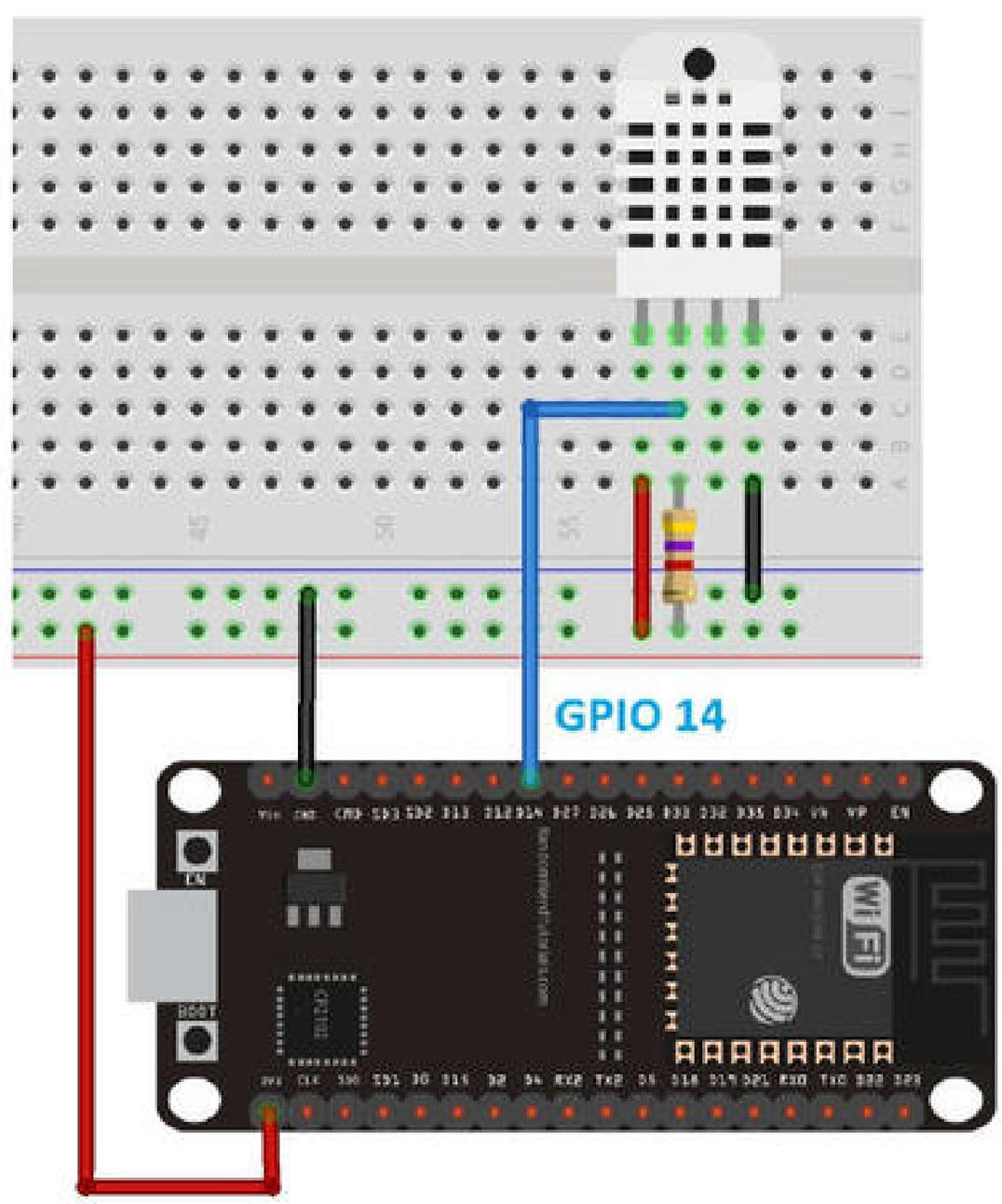
Parts required

Here's a list of parts you need to build the circuit:

- ESP32 or ESP8266
- DHT11 or DHT22 temperature and humidity sensor
- 4.7k Ohm resistor
- Breadboard
- Jumper wires

Schematic – ESP32

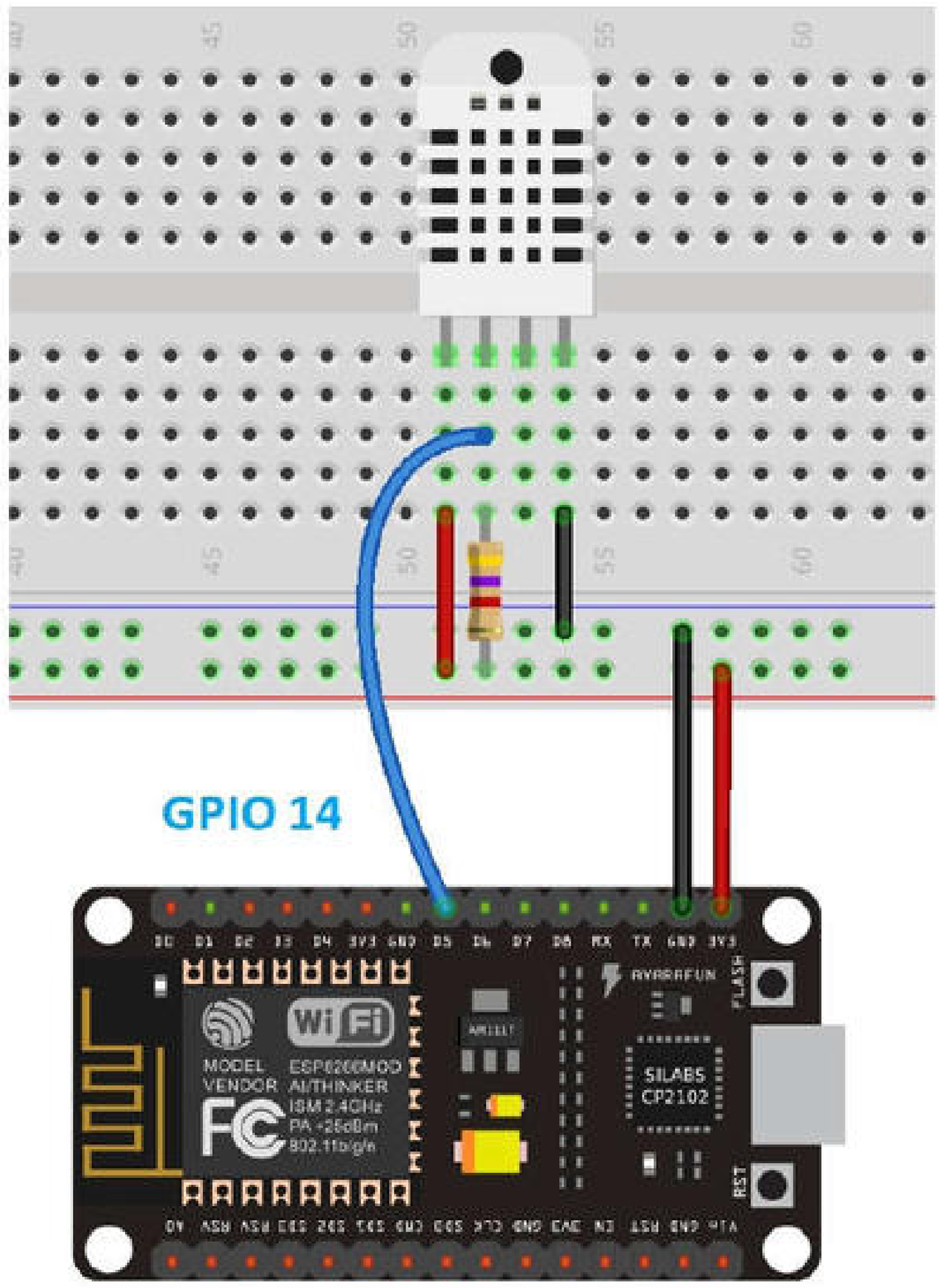
Follow the next schematic diagram if you're using an ESP32 board:



This schematic uses the ESP32 DEVKIT DOIT board version with 36 GPIOs. Before assembling the circuit double-check the pinout for the board you're using.

Schematic – ESP8266

Follow the next schematic diagram if you're using an ESP8266 board:



boot.py

Copy the following code to your *boot.py* file to enable Wi-Fi and connect the ESP board to your local network.

```
try:
  import usocket as socket
except:
  import socket

import network
from machine import Pin
import dht

import esp
esp.osdebug(None)

import gc
```

```
gc.collect()

ssid = 'REPLACE_WITH_YOUR_SSID'
password = 'REPLACE_WITH_YOUR_PASSWORD'

station = network.WLAN(network.STA_IF)

station.active(True)
station.connect(ssid, password)

while station.isconnected() == False:
  pass

print('Connection successful')
print(station.ifconfig())

sensor = dht.DHT22(Pin(14))
#sensor = dht.DHT11(Pin(14))
```

SOURCE CODE
https://github.com/RuiSantosdotme/ESP-MicroPython/blob/master/code/WiFi/Web_Server_DHT/boot.py

Besides the usual modules, you need to import the `dht` module to be able to read from the DHT sensor:

```
import dht
```

You also need to initialize the sensor by creating a `dht` instance on GPIO 14 as follows:

```
sensor = dht.DHT22(Pin(14))
```

If you're using a DHT11 sensor, uncomment the next line, and comment the previous one:

```
#sensor = dht.DHT11(Pin(14))
```

Don't forget to add your network credentials on the following lines:

```
ssid = 'REPLACE_WITH_YOUR_SSID'
password = 'REPLACE_WITH_YOUR_PASSWORD'
```

main.py

Copy the following code to your *main.py* file.

```
def read_sensor():
  global temp, temp_percentage, hum
  temp = temp_percentage = hum = 0
  try:
    sensor.measure()
    temp = sensor.temperature()
    hum = sensor.humidity()
    if (isinstance(temp, float) and isinstance(hum, float)) or
(isinstance(temp, int) and isinstance(hum,int)):
      msg = (b'{0:3.1f},{1:3.1f}'.format(temp, hum))

      temp_percentage = (temp+6)/(40+6)*(100)
      # uncomment for Fahrenheit
      #temp = temp * (9/5) + 32.0
      #temp_percentage = (temp-21)/(104-21)*(100)

      hum = round(hum, 2)
      return(msg)
    else:
      return('Invalid sensor readings.')
  except OSError as e:
    return('Failed to read sensor.')

def web_page():
  html = """<html><head><meta name="viewport"
content="width=device-width, initial-scale=1">
  <style>body{padding: 20px; margin: auto; width: 50%; text-
align: center;}
  .progress{background-color: #F5F5F5;}
.progress.vertical{position: relative;
  width: 25%; height: 60%; display: inline-block; margin: 20px;}
  .progress.vertical > .progress-bar{width: 100%
!important;position: absolute;bottom: 0;}
  .progress-bar{background: linear-gradient(to top, #f5af19 0%,
#f12711 100%);}
  .progress-bar-hum{background: linear-gradient(to top, #9CECFB
0%, #65C7F7 50%, #0052D4 100%);}
  p{position: absolute; font-size: 1.5rem; top: 50%; left: 50%;
transform: translate(-50%, -50%); z-index: 5;}</style></head>
  <body><h1>DHT Sensor</h1><div class="progress vertical">
  <p>"""+str(temp)+"""*<p>
```

```
  <div role="progressbar" style="height: """+str(temp_percentage)+"""%;" class="progress-bar"></div></div><div class="progress vertical">
  <p>"""+str(hum)+"""%</p>
  <div role="progressbar" style="height: """+str(hum)+"""%;" class="progress-bar progress-bar-hum"></div></div></body></html>"""
  return html

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind(('', 80))
s.listen(5)

while True:
  conn, addr = s.accept()
  print('Got a connection from %s' % str(addr))
  request = conn.recv(1024)
  print('Content = %s' % str(request))
  sensor_readings = read_sensor()
  print(sensor_readings)
  response = web_page()
  conn.send('HTTP/1.1 200 OK\n')
  conn.send('Content-Type: text/html\n')
  conn.send('Connection: close\n\n')
  conn.sendall(response)
  conn.close()
```

SOURCE CODE
https://github.com/RuiSantosdotme/ESP-MicroPython/blob/master/code/WiFi/Web_Server_DHT/main.py

Reading the sensor

First, create a function called `read_sensor()` that reads temperature and humidity. You can use that function in any other projects in which you need to take sensor readings from DHT sensors.

```
def read_sensor():
```

The function starts by setting three gloabal variables, so we can use them in all parts of the script (they are initilazed as 0).

```
global temp, temp_percentage, hum
temp = temp_percentage = hum = 0
```

The `temp` variable holds the temperature read from the sensor and the `hum` holds the humidity read from the sensor. Finally, the `temp_percentage` will hold a temperature value in percentage to be displayed on the progress bar.

Next, use `try` and `except` statements. Inside the `try` statement we try to get the temperature and humidity values.

Note: `try` and `except` allows us to continue the execution of the program when an exception happens. For example, when an error occurs, a `try` block code execution is stopped and transferred to the `except` block. In our example, the exception is especially useful to prevent the web server from crashing when we are not able to read from the sensor.

We measure the sensor by using the `measure()` method on the `sensor` object.

```
try:
  sensor.measure()
```

Then, read the temperature with `sensor.temperature()` and the humidity with `sensor.humidity()`. Save those readings on the `temp` and `hum` variables.

```
temp = sensor.temperature()
hum = sensor.humidity()
```

Valid temperature and humidity readings should be of type *float* (if you're using a DHT22 sensor) or type *int* (if you're using a DHT11) sensor. So, we check if we have valid readings by using the `isinstance()` function before proceeding.

```
if (isinstance(temp, float) and isinstance(hum, float)) or
(isinstance(temp, int) and isinstance(hum,int)):
```

Note: the `isinstance()` function accepts as arguments the variable and the data type: `isinstance(variable, data type)`

It returns `True` if the variable corresponds to the inserted data type and `False` if it doesn't.

If the readings are valid, prepare a message to be printed on the Shell that includes the temperature and humidity readings:

```
msg = (b'{0:3.1f},{1:3.1f}'.format(temp, hum))
```

After that, convert the temperature into a percentage value. We are assuming that 0% corresponds to -6°C and 100% to 40°C. You may want to change the range for your location if needed.

```
temp_percentage = (temp+6)/(40+6)*(100)
```

If you want to display temperature in Fahrenheit, uncomment the following lines and comment the previous one.

```
#temp = temp * (9/5) + 32.0
#temp_percentage = (temp-21)/(104-21)*(100)
```

Next, round the humidity to two decimal places and return the message:

```
hum = round(hum, 2)
return(msg)
```

In case we don't get valid sensor readings (not float type), we return '`Invalid sensor readings.`' message.

```
else:
  return('Invalid sensor readings.')
```

In case we're not able to read from the sensor (for example, in case it disconnects), return an error message.

```
except OSError as e:
  return('Failed to read sensor.')
```

That's how the function to read the sensor works. You can use this function in other projects that require sensor readings with the DHT11/DHT22 temperature and humidity sensor.

Web page

The `web_page()` function returns the HTML page. If we rearrange the HTML text, we have the following:

```
<html><head>
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <style>
    body {
      padding: 20px;
      margin: auto;
      width: 50%;
      text-align: center;
    }
    .progress { background-color: #F5F5F5; }
    .progress.vertical {
      position: relative;
      width: 25%;
      height: 60%;
      display: inline-block;
      margin: 20px;
    }
    .progress.vertical > .progress-bar {
      width: 100% !important;
      position: absolute;bottom: 0;
    }
    .progress-bar { background: linear-gradient(to top, #f5af19
0%, #f12711 100%); }
    .progress-bar-hum  {  background:  linear-gradient(to  top,
#9CECFB 0%, #65C7F7 50%, #0052D4 100%); }
    p {
      position: absolute;
      font-size: 1.5rem;
      top: 50%;
      left: 50%;
      transform: translate(-50%, -50%);
      z-index: 5;
    }
  </style>
</head><body>
  <h1>DHT Sensor</h1>
  <div class="progress vertical">
    <p>"""+str(temp)+"""*<p>
    <div role="progressbar" style="height:
"""+str(temp_percentage)+"""%;" class="progress-bar"></div>
  </div>
```

```
  <div class="progress vertical">
    <p>"""+str(hum)+"""%</p>
    <div role="progressbar" style="height: """+str(hum)+"""%;"
class="progress-bar progress-bar-hum"></div>
  </div>
</body></html>
```

SOURCE FILE
https://github.com/RuiSantosdotme/ESP-MicroPython/blob/master/code/WiFi/Web_Server_DHT/web_page.html

The text highlighted in red between the `<style>` and `</style>` tags is CSS to style the progress bar and the web page.

To display the sensor readings on the web page, concatenate the `html` string with the sensor readings. For example, the following line is a paragraph that displays the temperature reading:

```
<p>"""+str(temp)+"""*<p>
```

We need to convert the `temp` variable to a string type, so it can be concantenated with the `html` variable.

In the following line, set the height of the progress bar to `temp_percentage` variable value. The `height` attribute accepts a value between 0 and 100%.

```
<div role="progressbar" style="height:
"""+str(temp_percentage)+"""%;" class="progress-bar"></div>
```

Repeat the same process to display the humidity with the `hum` variable. Remember that the `hum` variable holds the humidity value in percentage.

```
<p>"""+str(hum)+"""%</p>
<div  role="progressbar"  style="height:  """+str(hum)+"""%;"
class="progress-bar progress-bar-hum"></div>
```

In summary, to display sensor readings, we can concatenate the readings with the rest of the `html` variable string.

Creating the web server

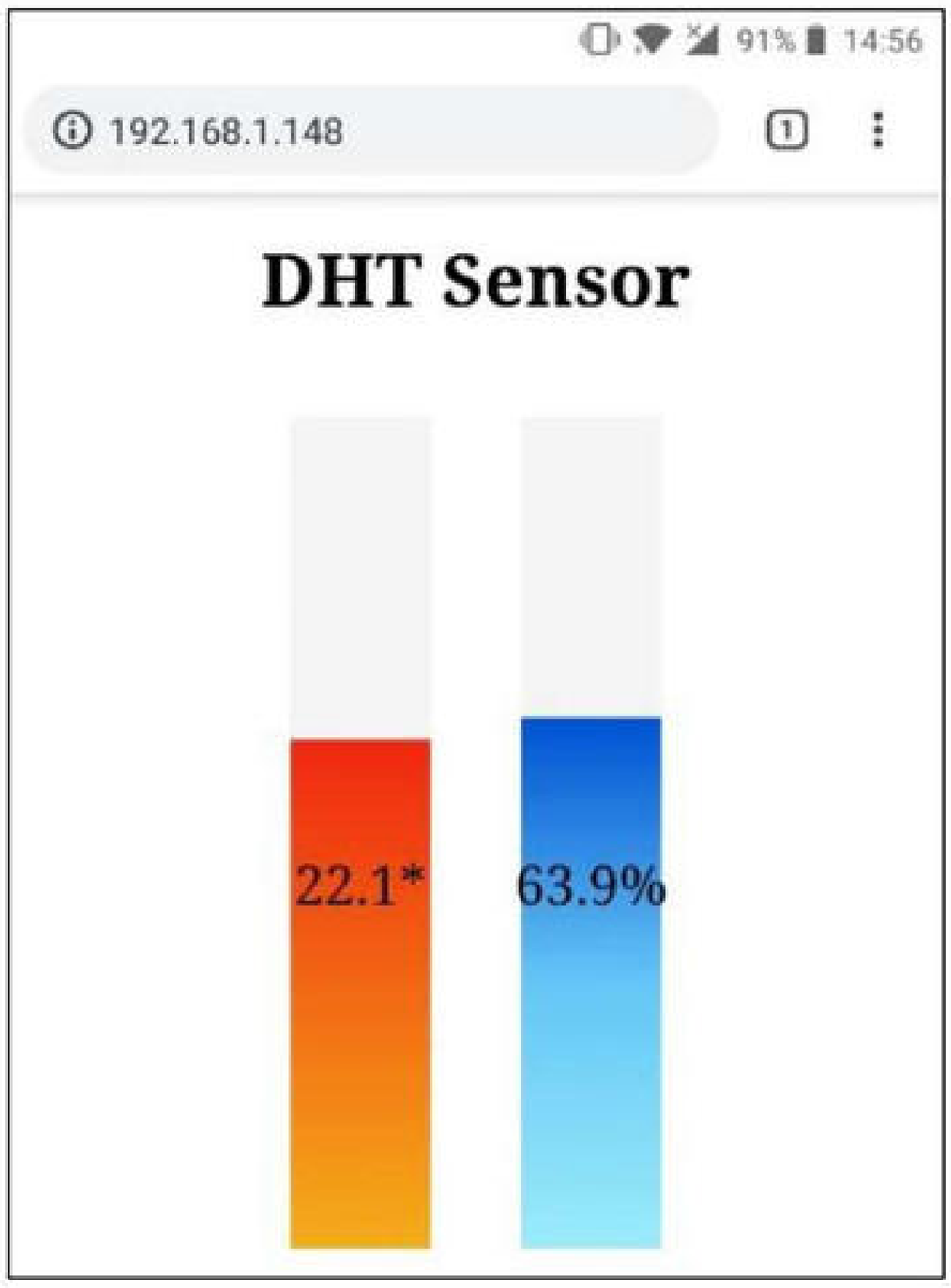
After that, make the usual procedures to create a socket server.

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind(('', 80))
s.listen(5)
while True:
  conn, addr = s.accept()
  print('Got a connection from %s' % str(addr))
  request = conn.recv(1024)
  print('Content = %s' % str(request))
  sensor_readings = read_sensor()
  print(sensor_readings)
  response = web_page()
  conn.send('HTTP/1.1 200 OK\n')
  conn.send('Content-Type: text/html\n')
  conn.send('Connection: close\n\n')
  conn.sendall(response)
  conn.close()
```

In the `while` loop, when we call the `read_sensor()` function to print the sensor readings, the global variables `temp`, `hum` and `temp_percentage` are updated. So, the `web_page()` function generates HTML text with the latest sensor readings.

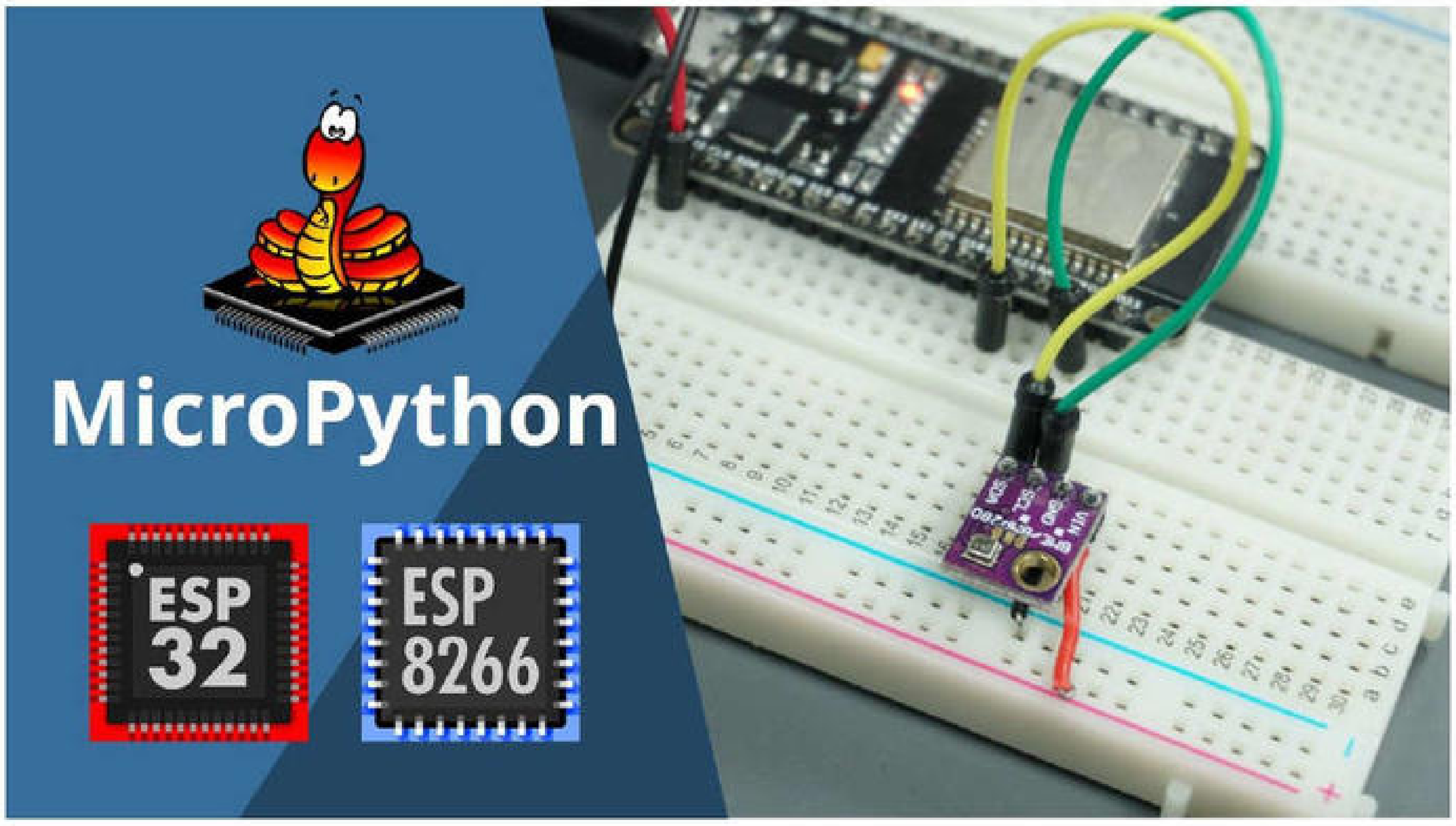
Demonstration

Upload the *boot.py* and *main.py* files to your ESP32/ESP8266. Open your browser and type your ESP IP address. You should get the latest sensor readings as shown in the figure below:



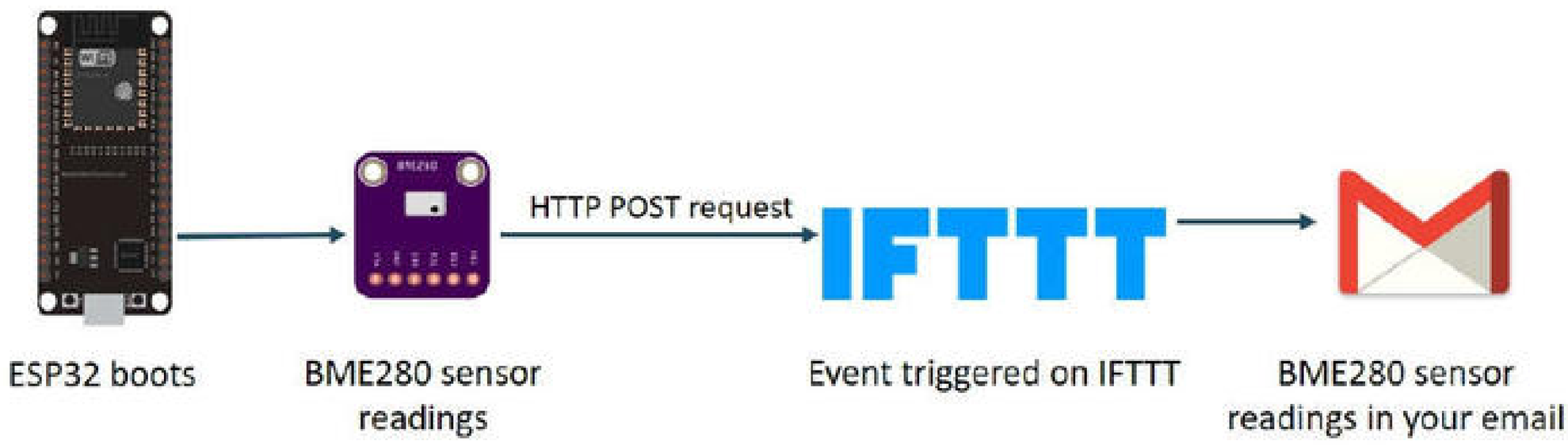
To update the sensor readings, you need to make a new request to the server - you need to refresh the web page.

Send Sensor Readings via Email (IFTTT)



In this project, we'll set up the ESP32/ESP8266 to send email notifications with temperature, humidity, and pressure readings. To get temperature, humidity, and pressure we use the BME280 sensor module. To send email notifications we use a third-party service called IFTTT. With this example, you'll learn how to read from the BME280 sensor and how to make HTTP POST requests.

Project Overview



Our example sends the sensor readings once on boot. This is a simple example that shows how to make HTTP POST requests. You can apply the code used in this project with other cloud services and in your own projects.

Introducing the BME280 Sensor Module

The BME280 sensor module reads temperature, humidity, and pressure. Because pressure changes with altitude, you can also estimate altitude. There are several versions of this sensor module, but we're using the one shown in the figure below.

The sensor communicates using I2C communication protocol.

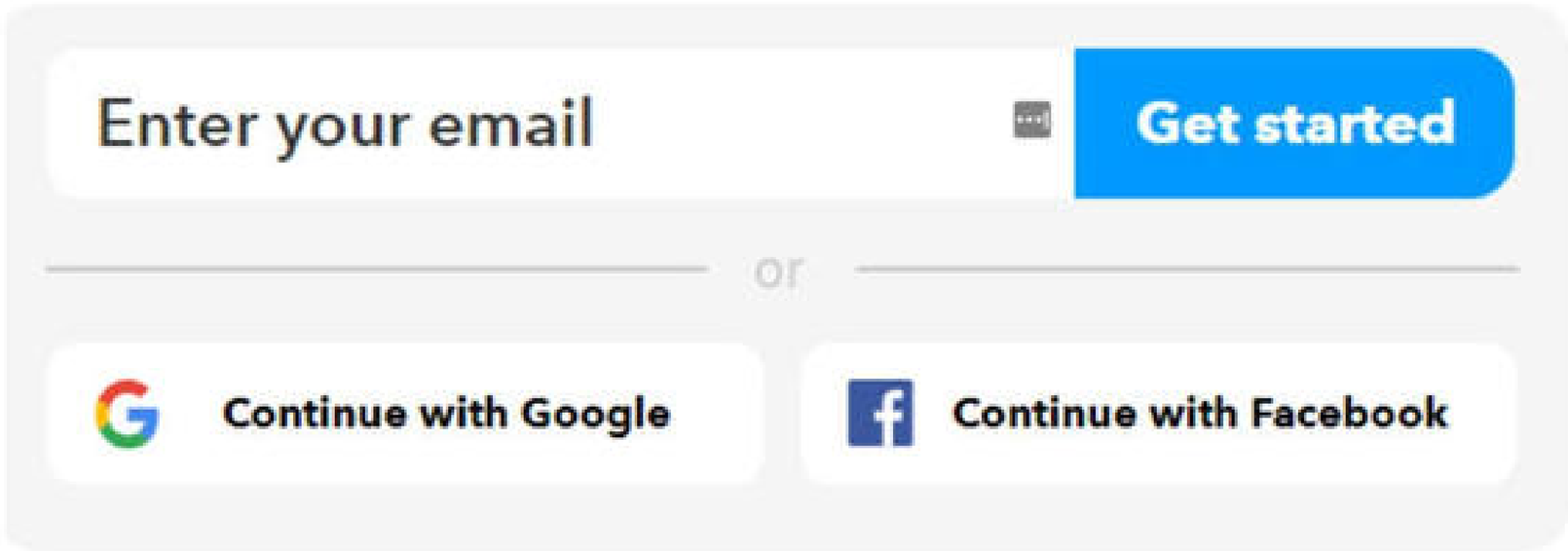
Introducing IFTTT

IFTTT stands for "If This Than That", and it is a free web-based service to create chains of simple conditional statements called applets.

This means you can trigger an event when something happens. In this example, the applet will send sensor readings to your email when the ESP32 or ESP8266 makes a request.

Creating an IFTTT Account

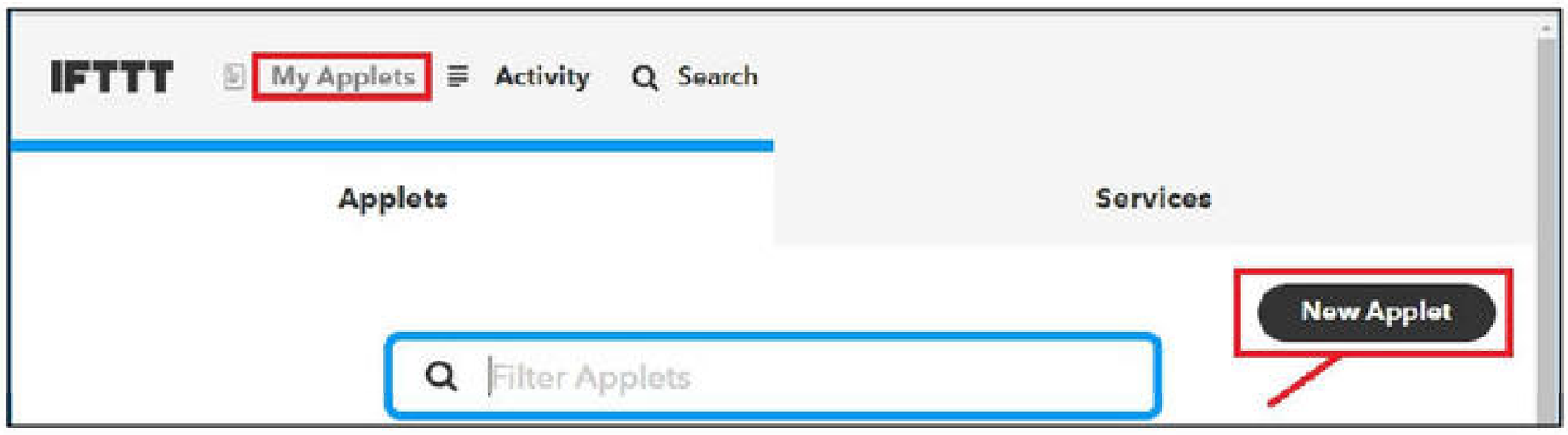
If you don't have an IFTTT account, go the official site: ifttt.com and enter your email to create an account and get started. Creating an account on IFTTT is free!



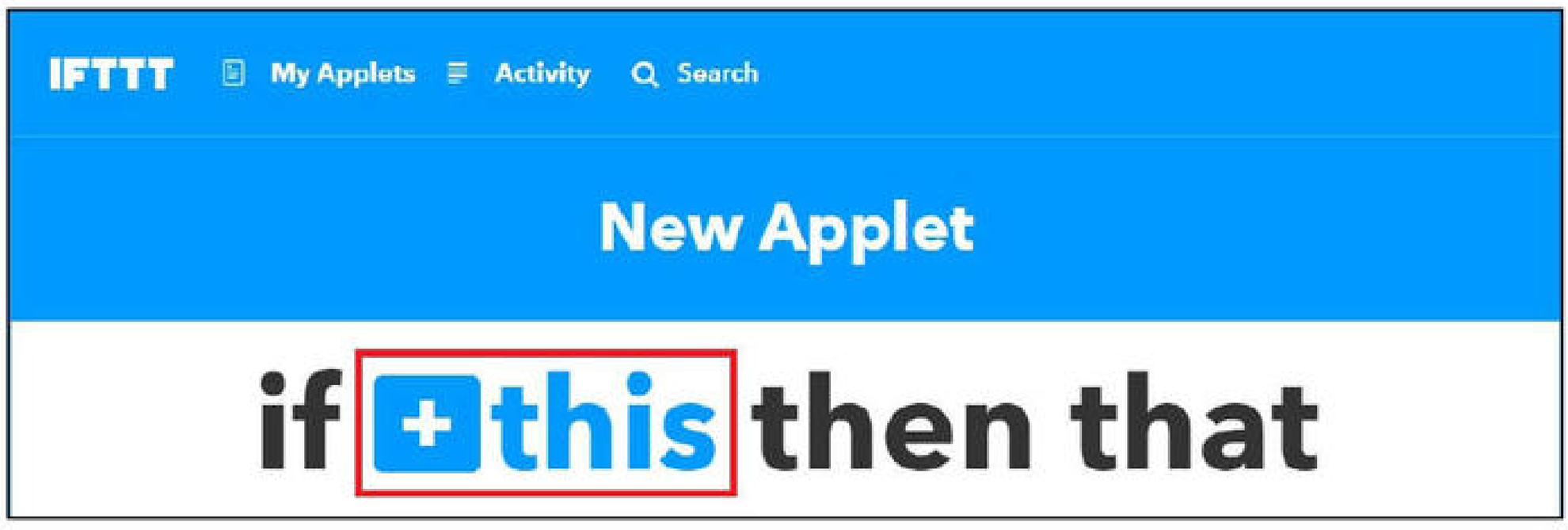
Creating an Applet

Next, you need to create a new applet. Follow the next steps to create a new applet:

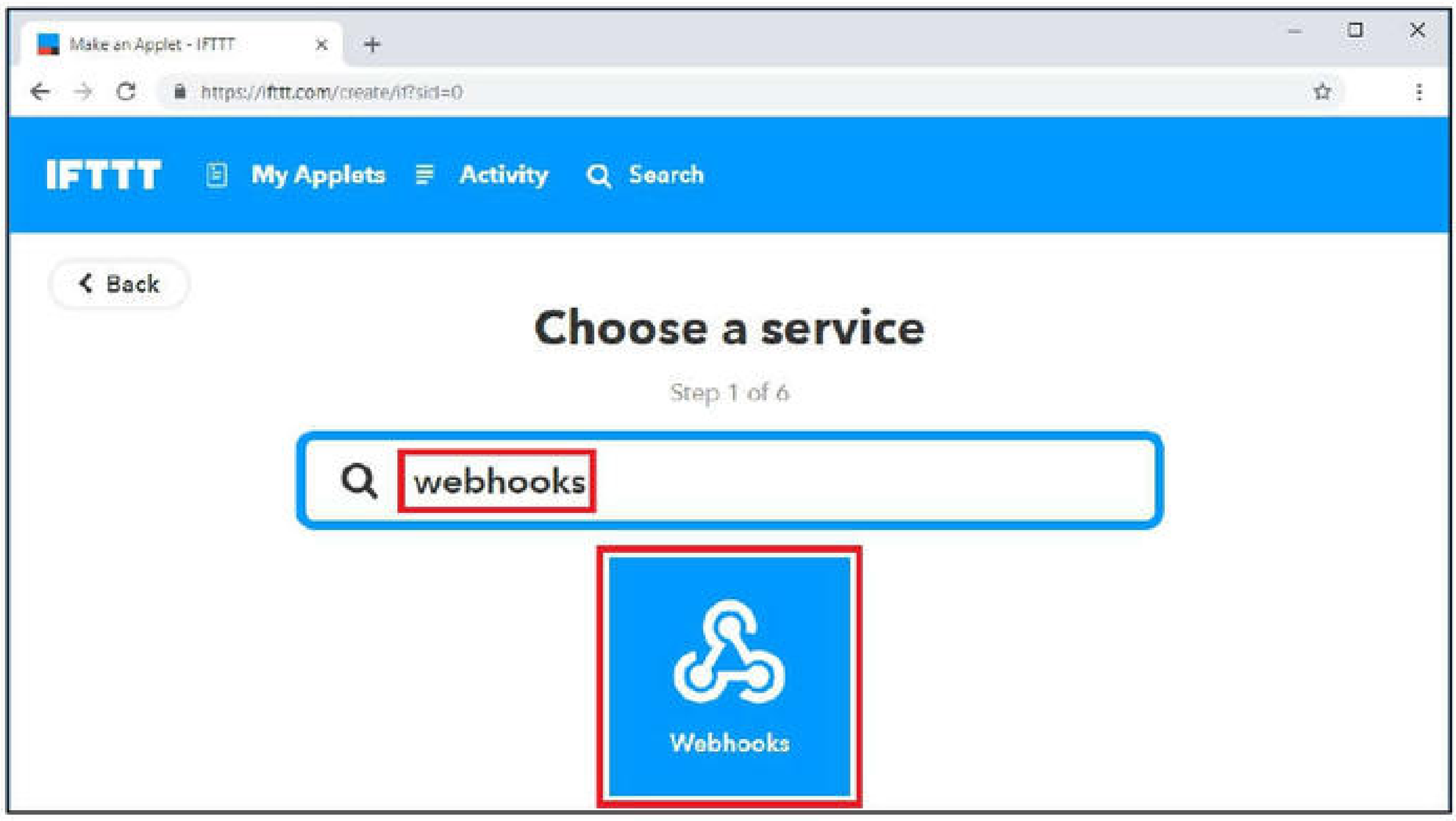
1) Go to "**My Applets**" and create a new applet by clicking the "**New Applet**" button.



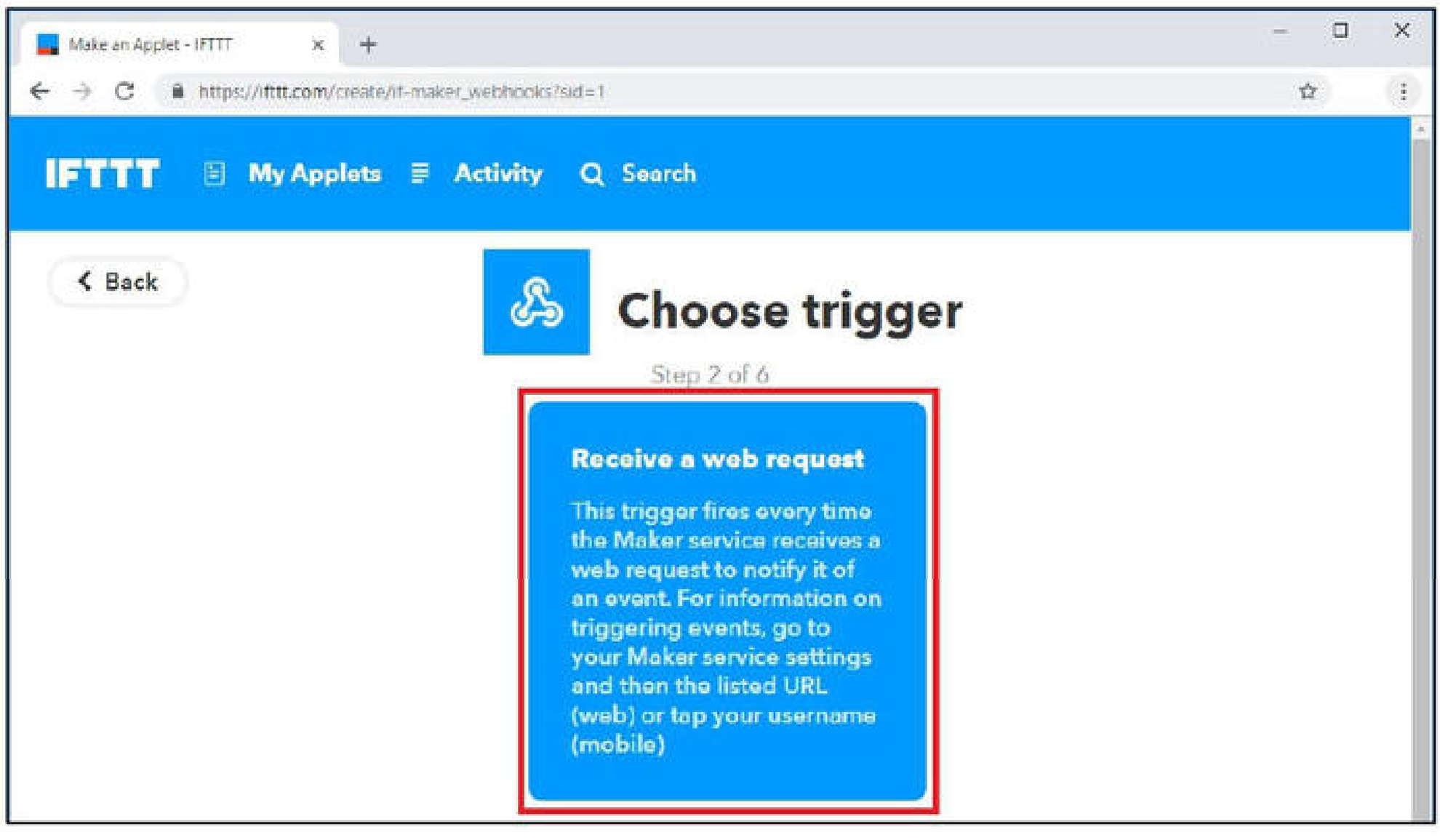
2) Click on the "**this**" word in a blue color – as highlighted in the figure below.



3) Search for the "**Webhooks**" service and select the **Webhooks** icon.

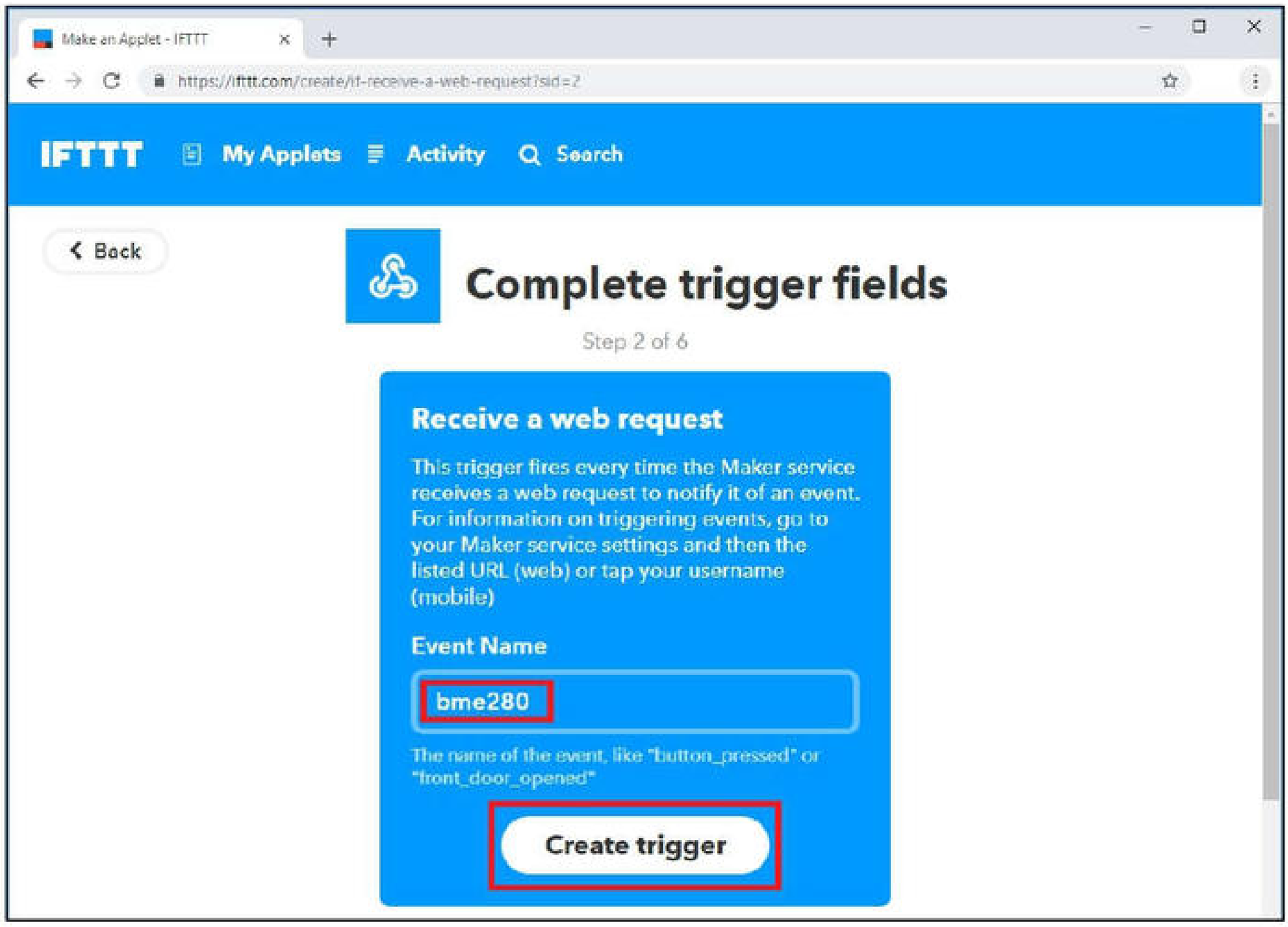


4) Choose the "**Receive a web request**" trigger.

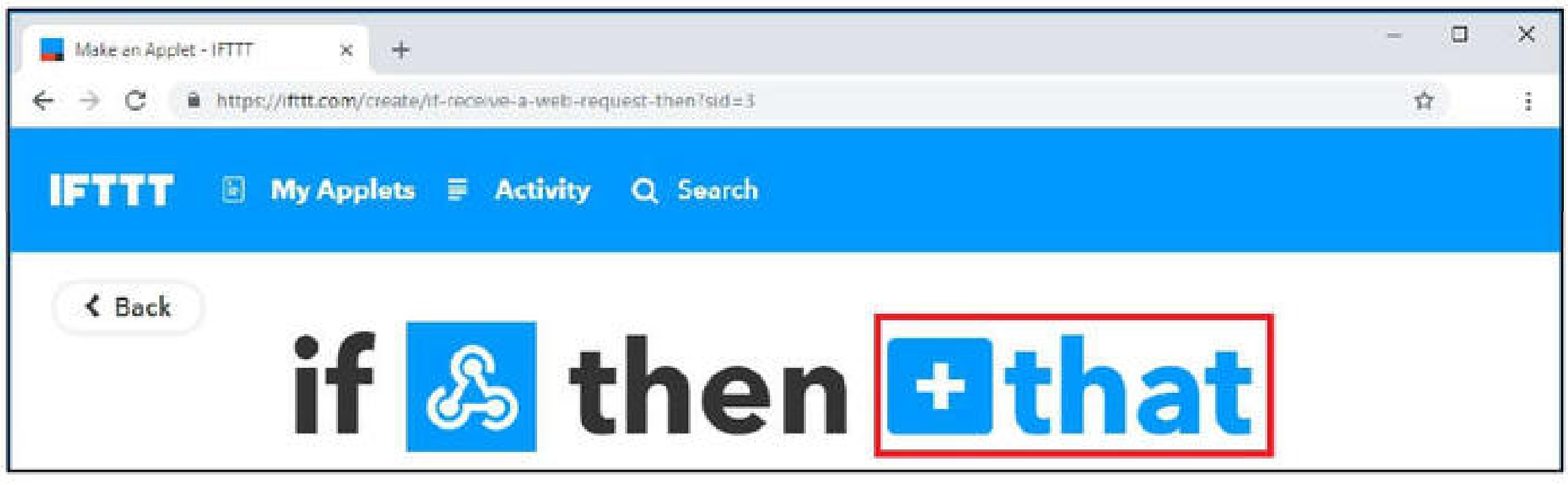


This request allows you to perform something when the IFTTT Maker Webhooks service receives a web request. We'll make a web request to the IFTTT Maker service with the ESP32/ESP8266.

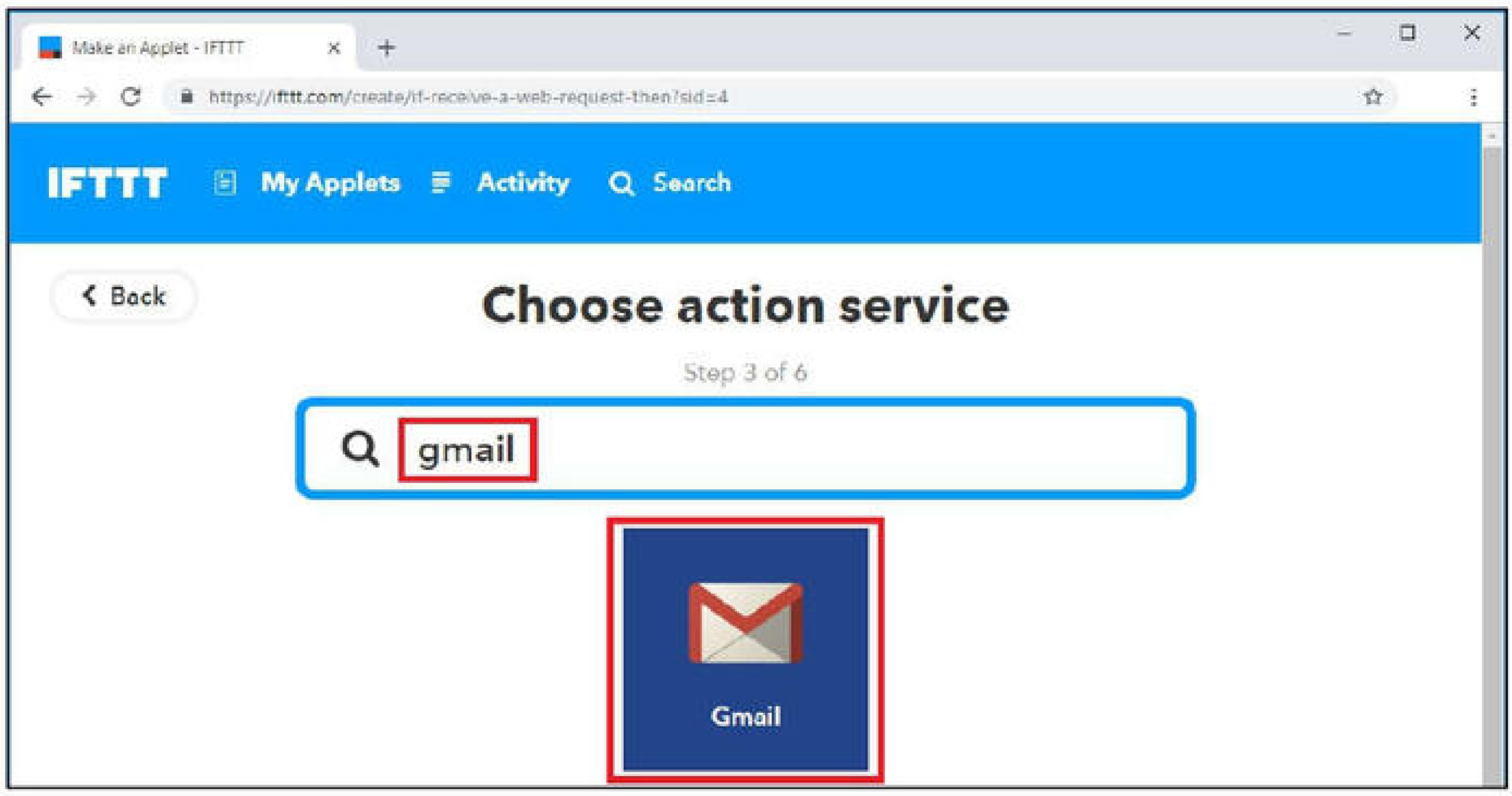
5) Give a name to the event. In this case "**bme280**" as shown in the figure below. Then, click the "**Create trigger**" button.



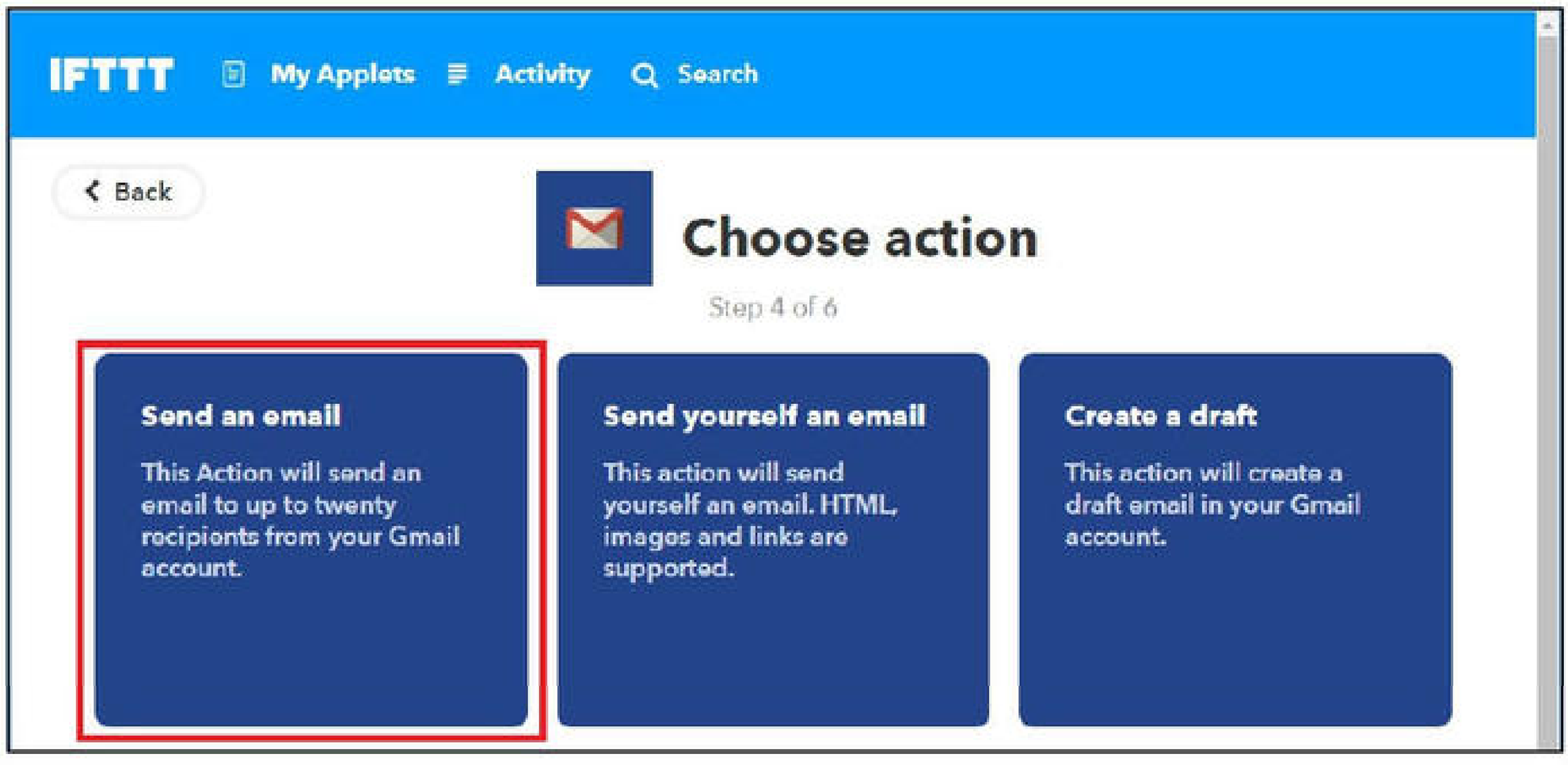
6) Click the "**that**" word to proceed. Now, we'll define what happens when the event you've defined is triggered.



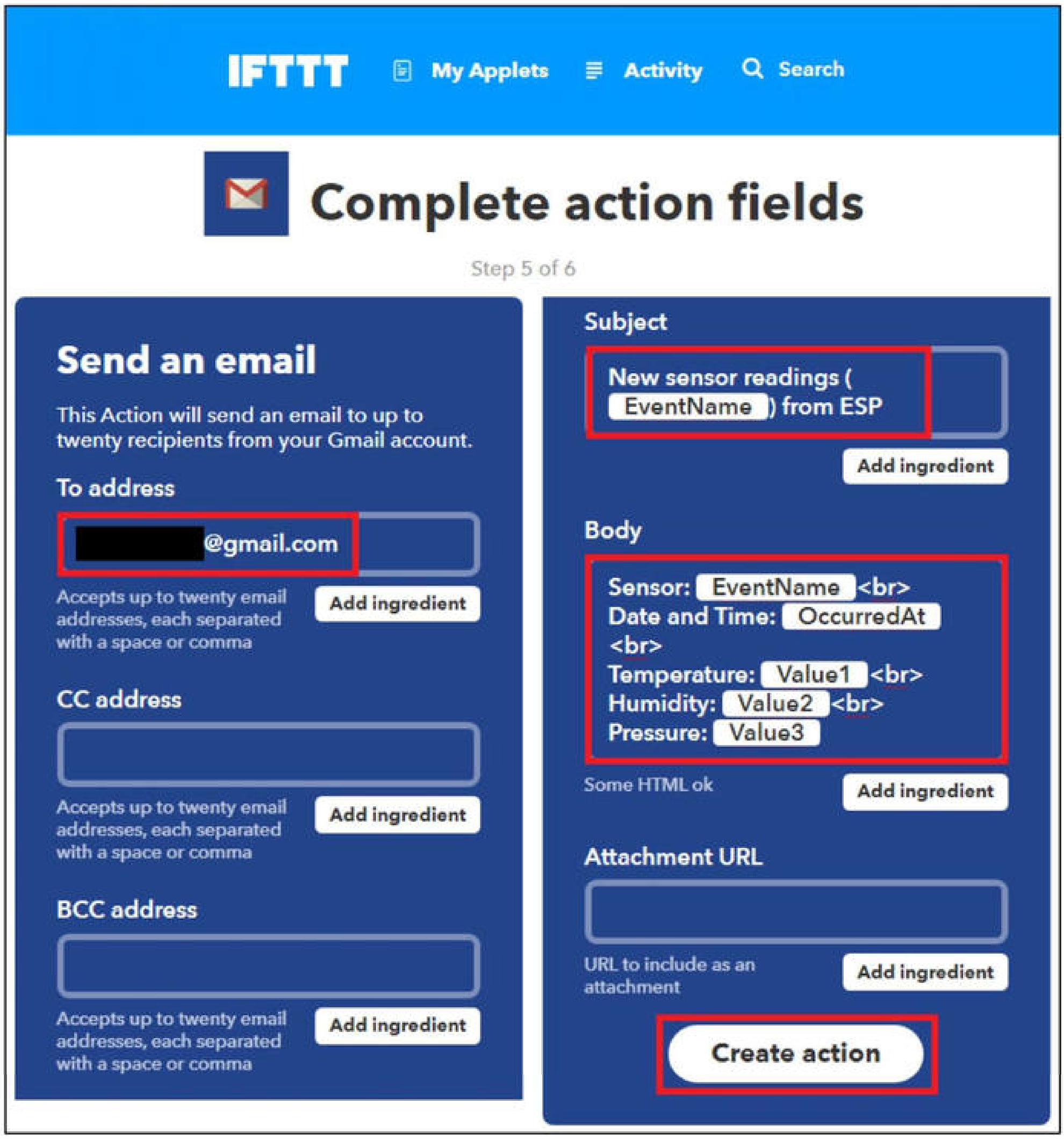
7) Search for the "**Gmail**" service and select the **Gmail** icon.



8) Next, select the "**Send an email**" option.

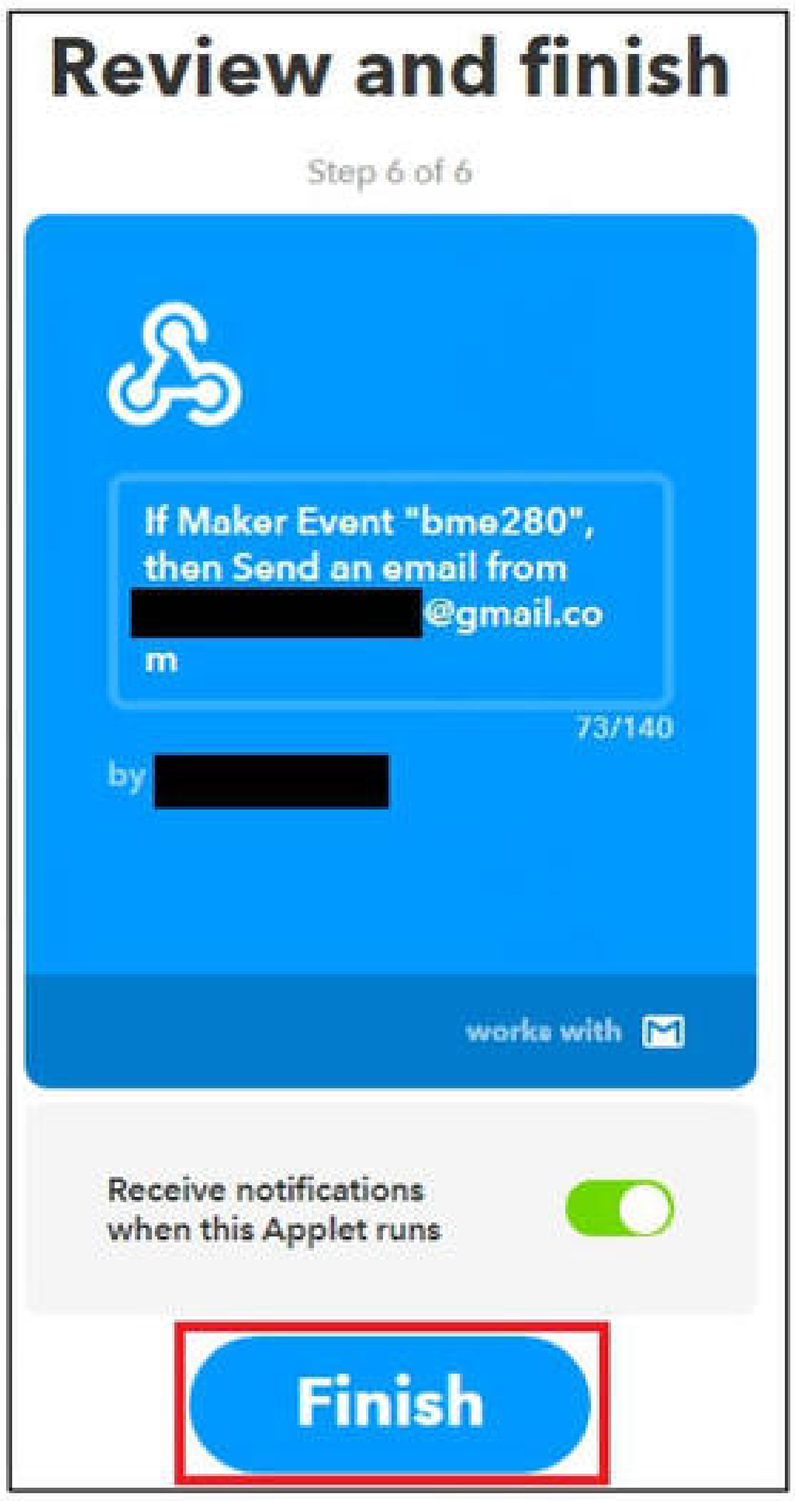


9) Then, complete the action fields. Enter your email in the "**To address**" field. Fill the "**Subject"** and "**Body"** fields as shown in the image below.

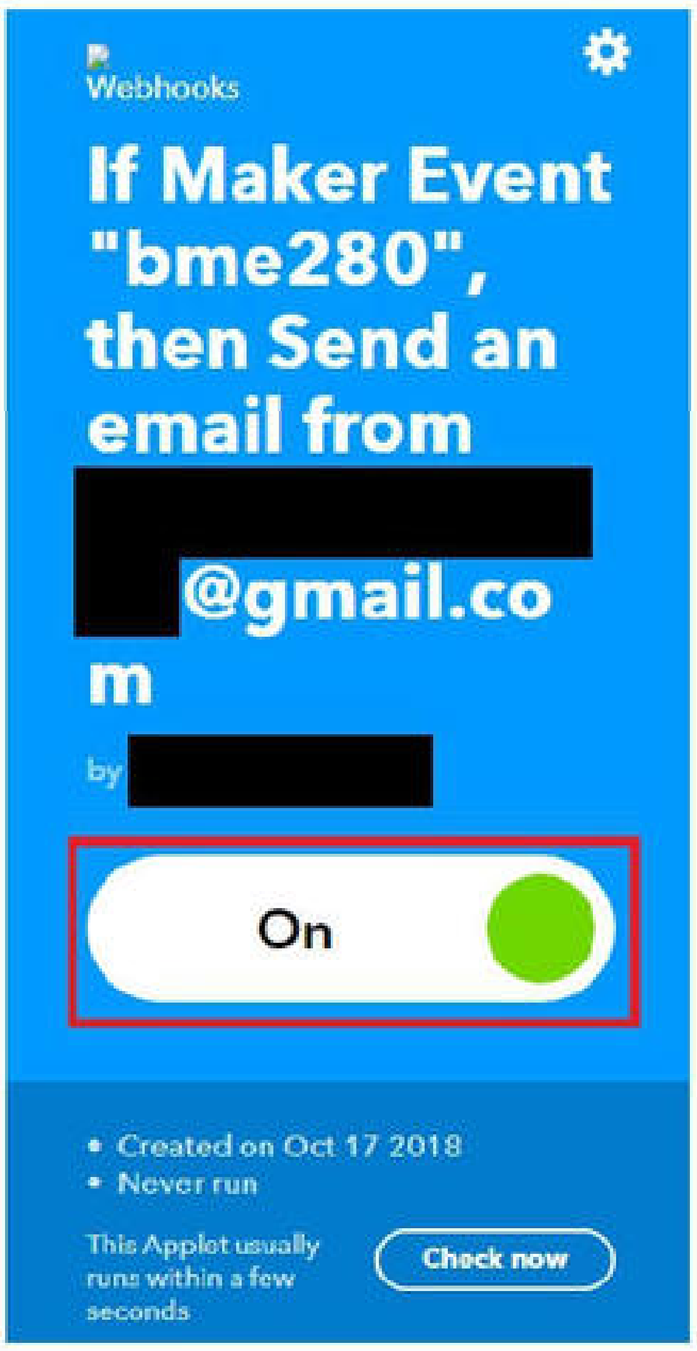


Finally, click the "**Create action**" button.

10) Press the "**Finish**" button to create your applet.



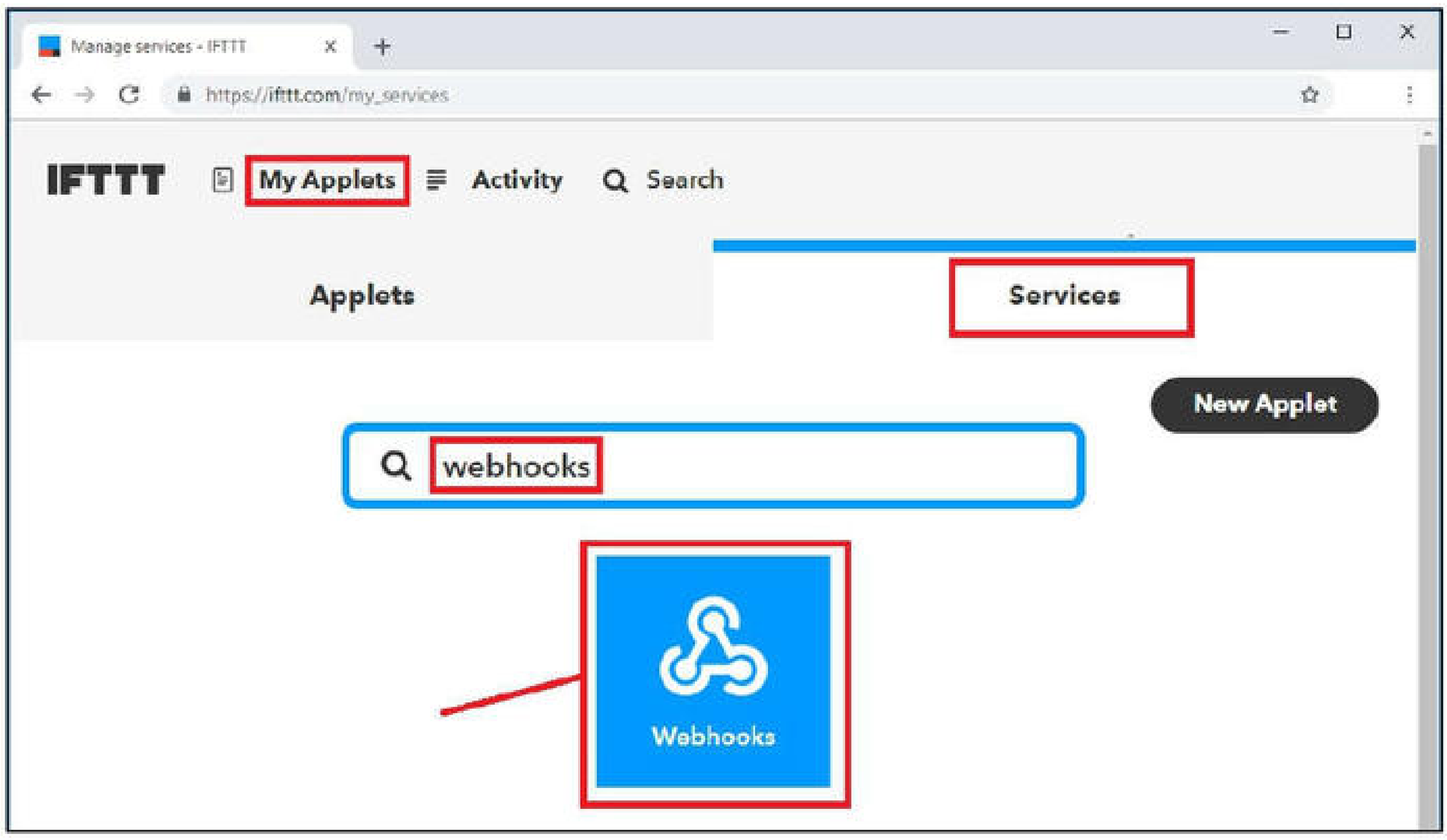
Then, go to **My Applets** and enable the Applet, as shown in the figure below:



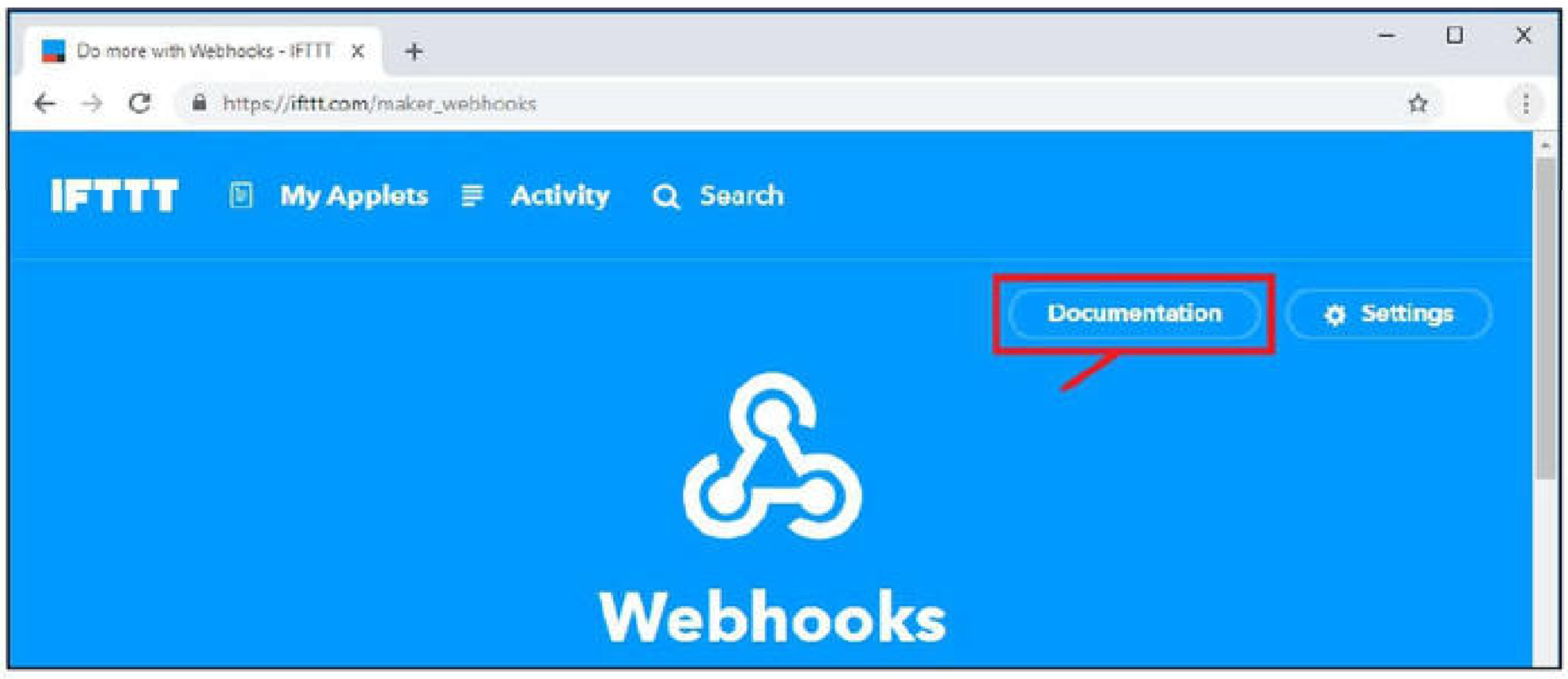
Testing Your Applet

Before proceeding with the project, it's very important to test your Applet first. Follow the next steps to test it:

1) Go to "**My Applets**", select the "**Services**" tab. Search for **Webhooks** and click the **Webhooks** icon.



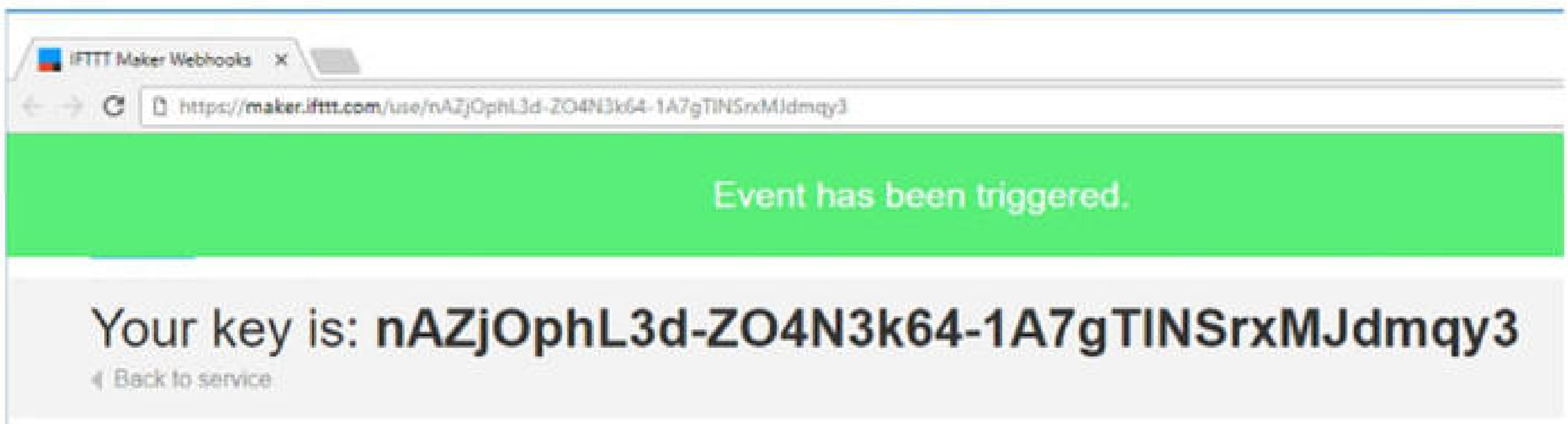
2) Click the "**Documentation**" button.



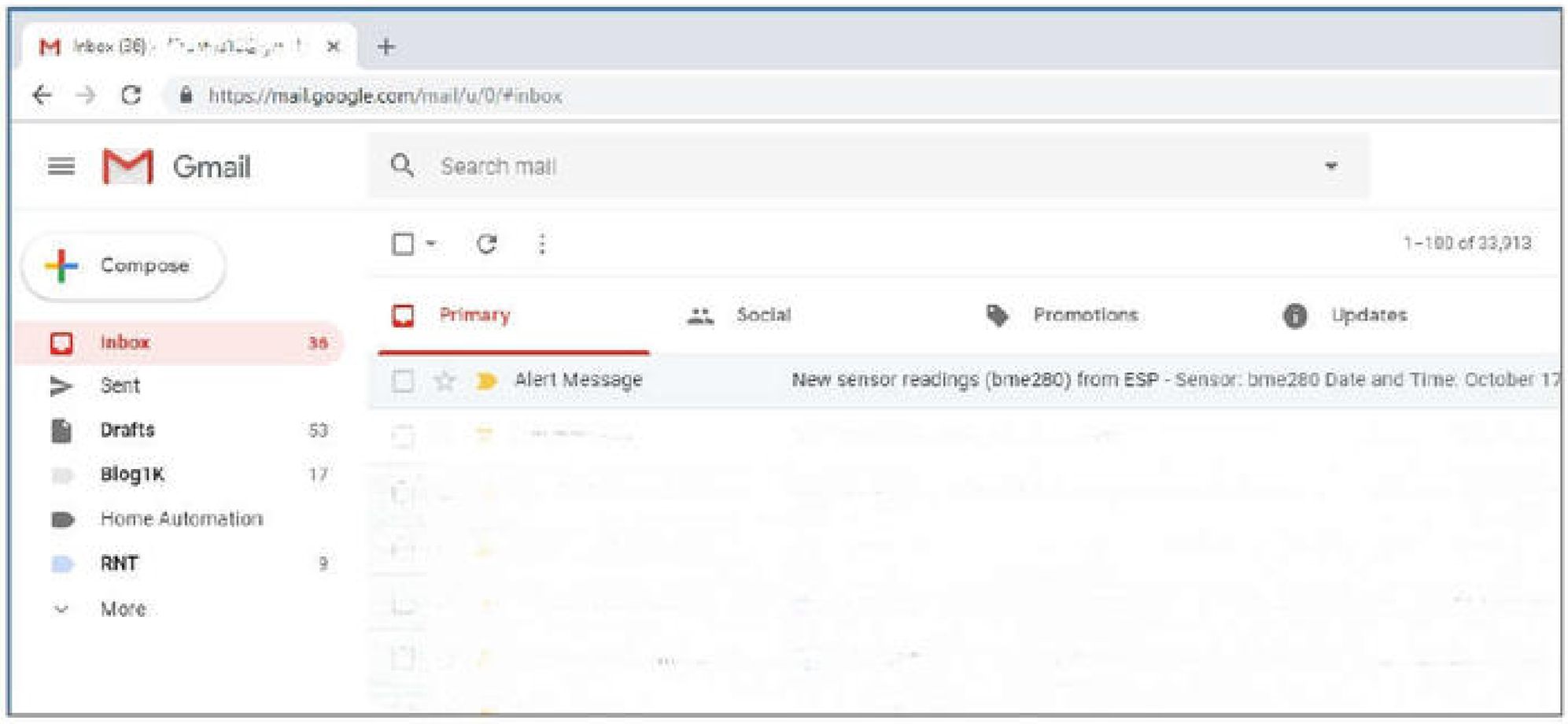
3) A page as shown in the following figure shows up.

The page shows your unique API key. You should not share your unique API key with anyone. Fill the "**To trigger an Event**" section as shown in the figure – it is highlighted with red rectangles. Then, click the "**Test it**" button.

4) The event should be successfully triggered, and you'll get a green message saying "**Event has been triggered**".



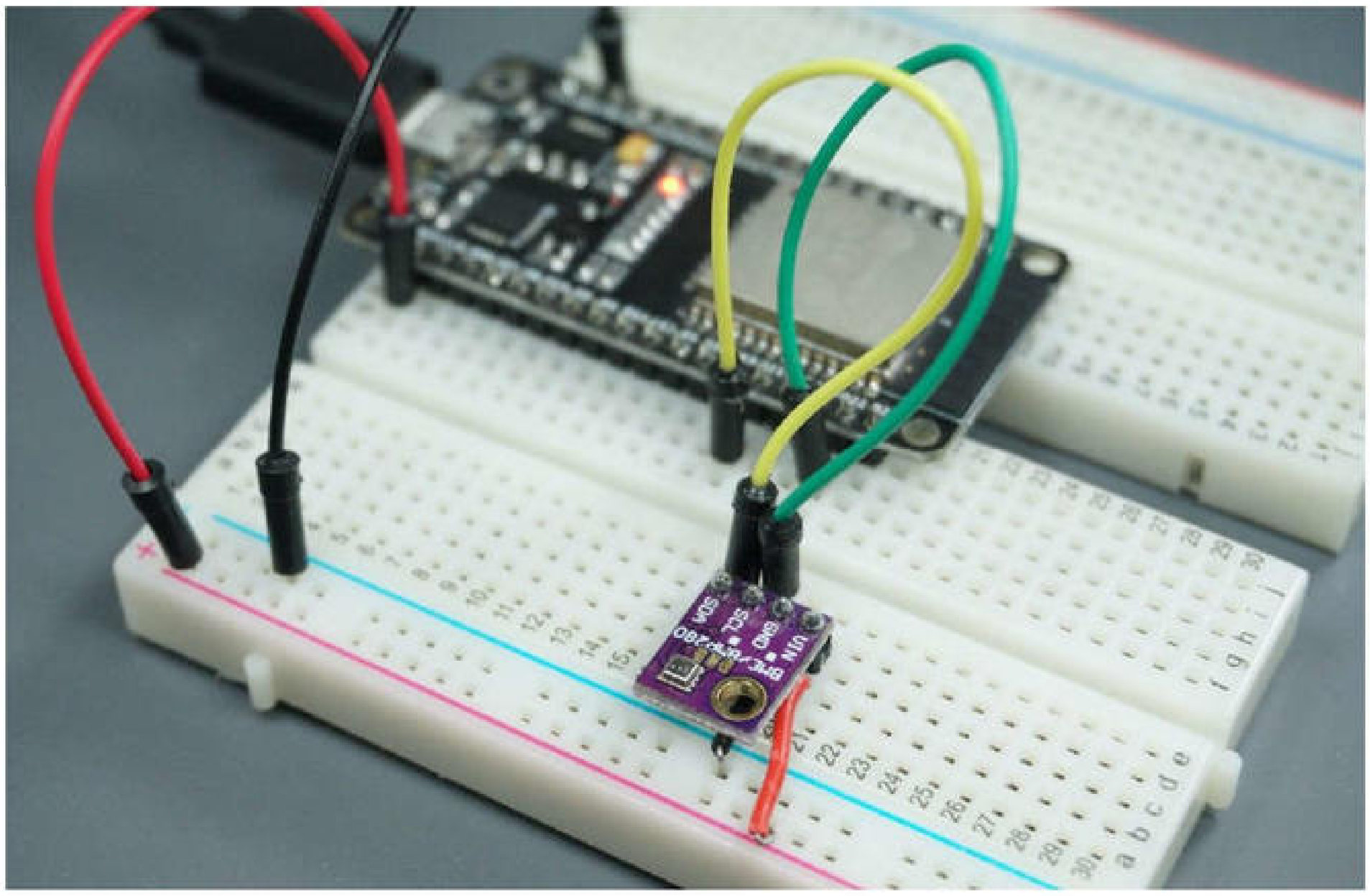
5) Go to your Gmail account. You should have a new email in your inbox from the IFTTT service with the values you've defined in the previous step:



If you've received an email with the data entered in the test request, it means your applet is working as expected. Now, we need to program the ESP32/ESP8266 to send an HTPP POST request to the IFTTT service with the sensor readings.

Schematic

For this project you need to wire the BME280 sensor module to the ESP32 or ESP8266 I2C pins.



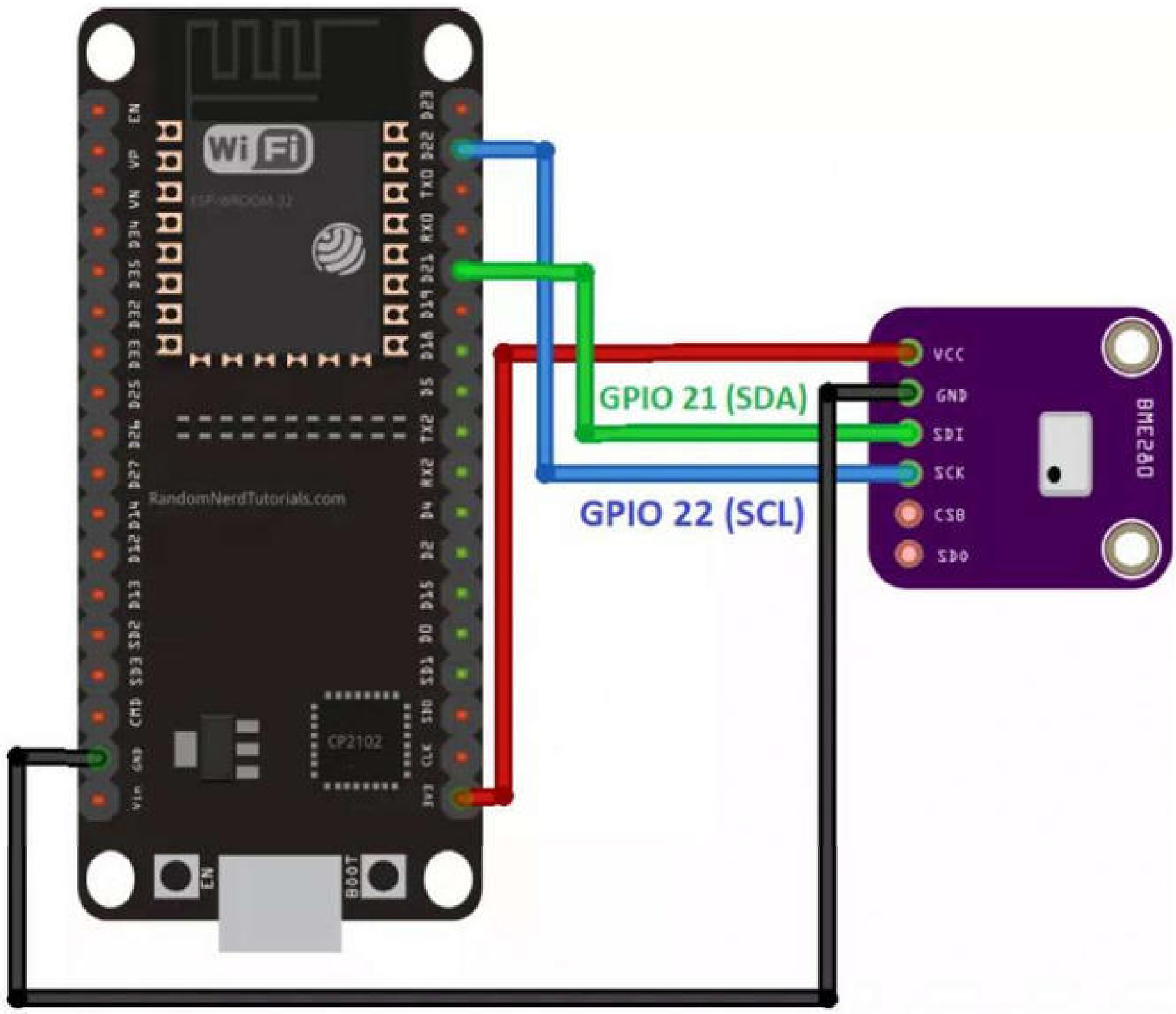
Parts required

Here's a list of parts you need to build the circuit:

- ESP32 or ESP8266
- BME280 sensor module
- Jumper wires

Schematic – ESP32

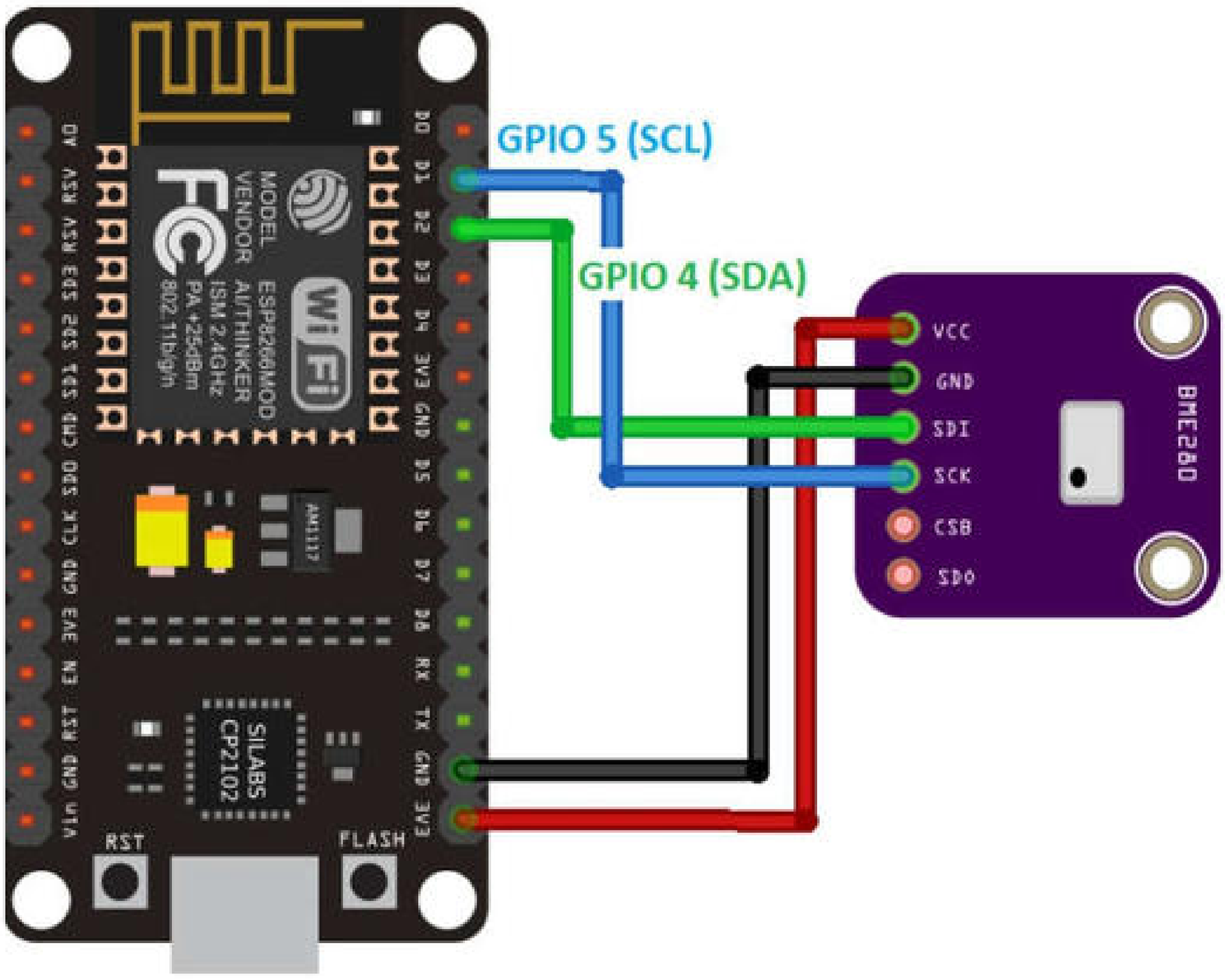
Follow the next schematic diagram if you're using an ESP32 board:



This schematic uses the ESP32 DEVKIT DOIT board version with 36 GPIOs. Before assembling the circuit double-check the pinout for the board you're using.

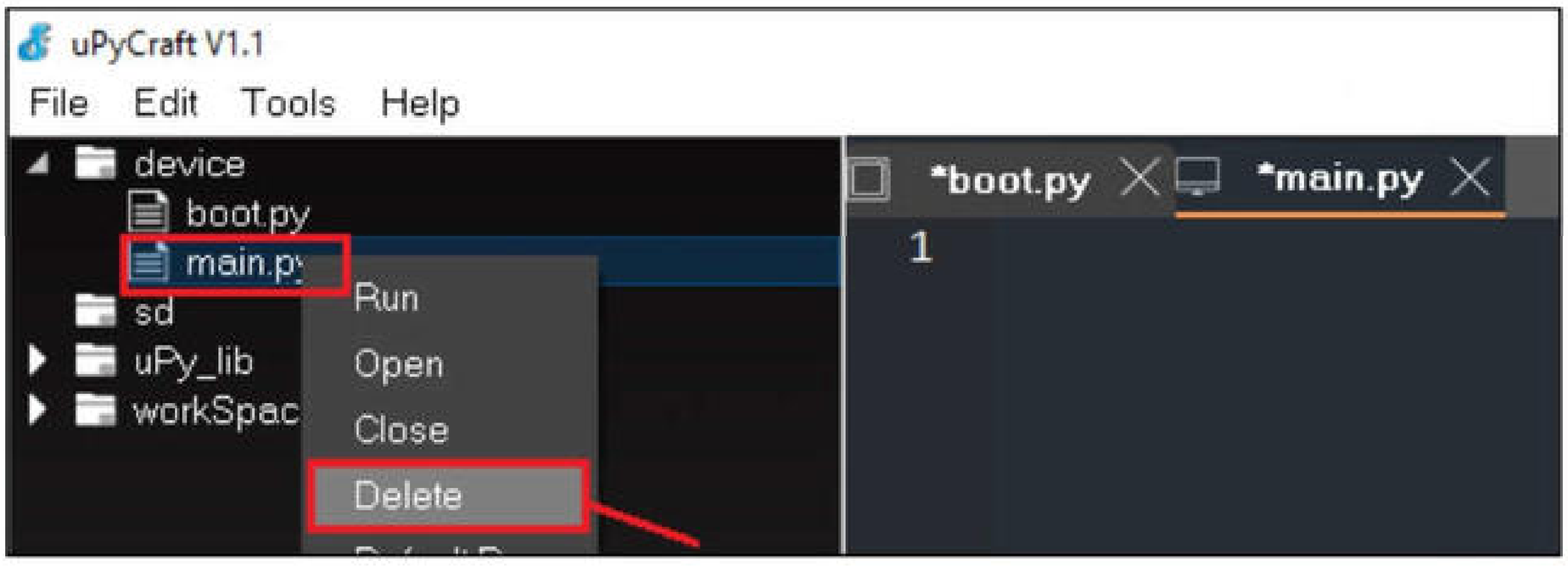
Schematic – ESP8266

Follow the next schematic diagram if you're using an ESP8266 board:



Preparing the Files

For this project we don't need the *main.py* file, so you can delete it. Open the *device* folder, right-click the *main.py* file and click **Delete**.



Press OK to confirm:

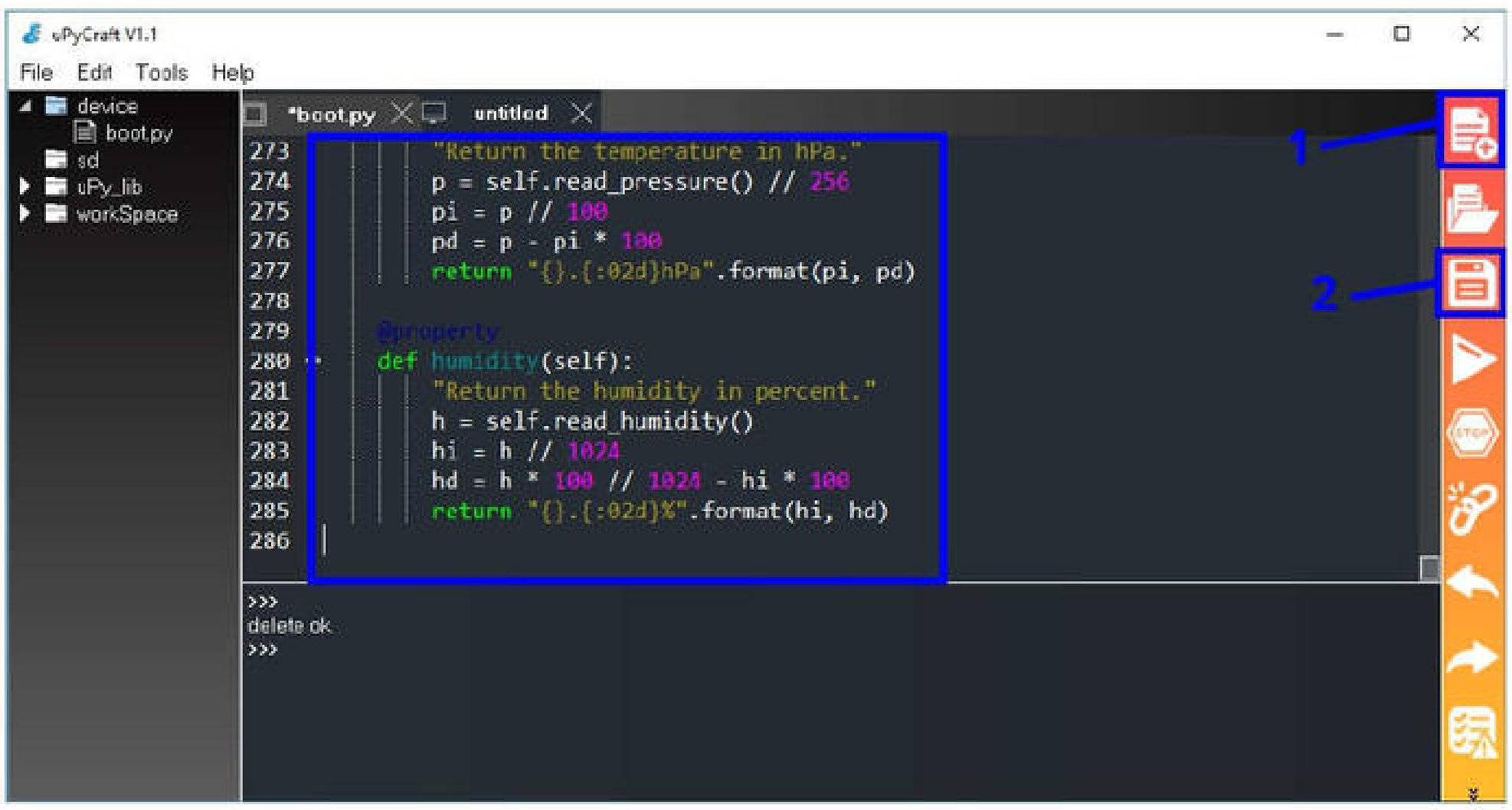
BME280 library

The library to read from the BME280 sensor isn't part of the standard MicroPython library by default. So, you need to upload the library to your ESP32/ESP8266 board.

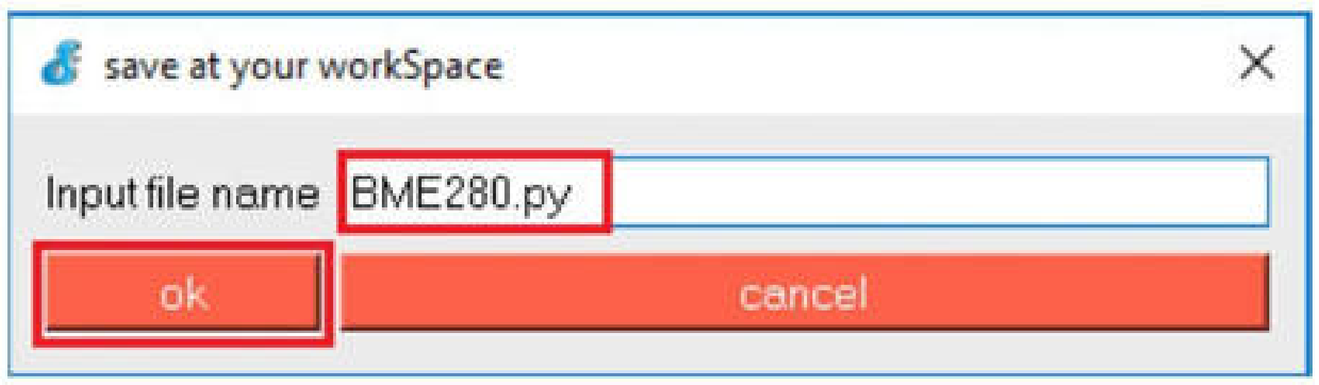
Create a new file by pressing the **New File** button (1). Copy the BME280 library code into it. The BME280 library code can be found in the following link:

https://github.com/RuiSantosdotme/ESP-MicroPython/blob/master/code/WiFi/HTTP_Client_IFTTT_BME280/BME280.py

Save the file by pressing the **Save** button (2).

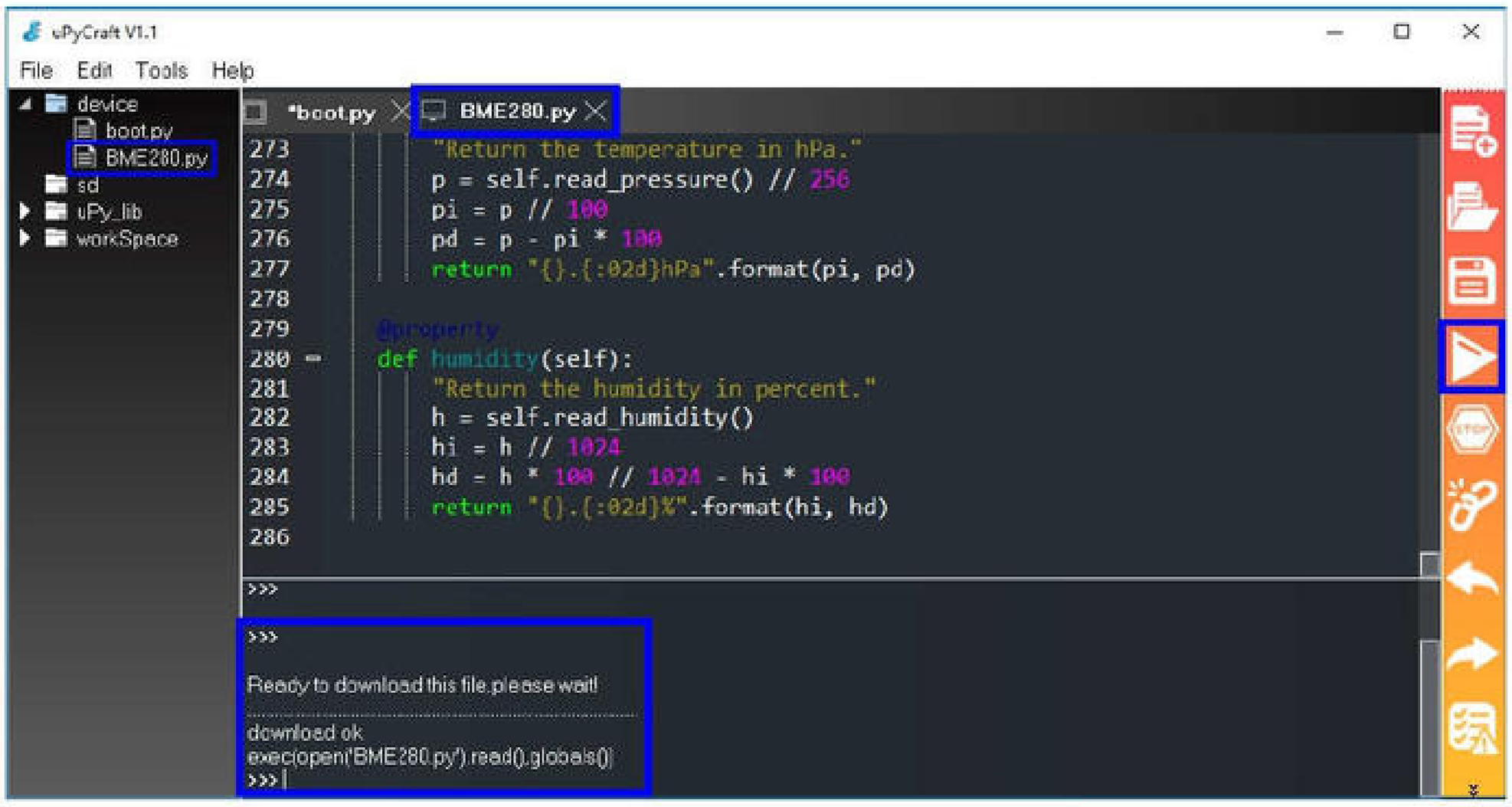


Call this new file "**BME280.py**" and press **ok**.



Click the **Download and Run** button.

The file should be saved on the *device* folder with the name "**BME280.py**" as highlighted in the figure below.



Now, you can use the library functionalities in your code by importing the library.

Note: if you're using Thonny IDE, you just need to:

1. Copy the library code to a new file.

2. Name that file *BME280.py*.

3. Go to **Device ▸ Upload current script with the current name**.

boot.py

Copy the following code to your *boot.py* file.

```
from machine import Pin, I2C
import BME280
import network
import urequests

import esp
esp.osdebug(None)
import gc
gc.collect()

ssid = 'REPLACE_WITH_YOUR_SSID'
password = 'REPLACE_WITH_YOUR_PASSWORD'

api_key = 'REPLACE_WITH_YOUR_WEBHOOKS_IFTTT_API_KEY'

station = network.WLAN(network.STA_IF)
```

```
station.active(True)
station.connect(ssid, password)

while station.isconnected() == False:
  pass

print('Connection successful')
print(station.ifconfig())

# ESP32 - Pin assignment
i2c = I2C(scl=Pin(22), sda=Pin(21), freq=10000)
# ESP8266 - Pin assignment
#i2c = I2C(scl=Pin(5), sda=Pin(4), freq=10000)

try:
  bme = BME280.BME280(i2c=i2c)
  temp = bme.temperature
  hum = bme.humidity
  pres = bme.pressure

  # uncomment for temperature in Fahrenheit
  #temp = (bme.read_temperature()/100) * (9/5) + 32
  #temp = str(round(temp, 2)) + 'F'

  sensor_readings = {'value1':temp, 'value2':hum, 'value3':pres}
  print(sensor_readings)

  request_headers = {'Content-Type': 'application/json'}

  request = urequests.post(
    'http://maker.ifttt.com/trigger/bme280/with/key/' + api_key,
    json=sensor_readings,
    headers=request_headers)
  print(request.text)
  request.close()

except OSError as e:
  print('Failed to read/publish sensor readings.')
```

SOURCE CODE

https://github.com/RuiSantosdotme/ESP-MicroPython/blob/master/code/WiFi/HTTP_Client_IFTTT_BME280/boot.py

How the Code Works

You start by importing the necessary modules. To interact with the GPIOs and read data from the sensor via I2C communication, you need to import the `Pin` and `I2C` classes from the `machine` module.

```
from machine import Pin, I2C
```

You also need to import the `BME280` library that you previously uploaded to the board as *BME280.py* file.

```
import BME280
```

To connect to your network, you need to import the `network` module.

```
import network
```

Finally, you also need the `urequests` library. This library allows you to make HTTP requests. In this example, we'll make a request to the IFTTT services to trigger the event we've created earlier.

```
import urequests
```

You need to include your SSID and password in the following variables:

```
ssid = 'REPLACE_WITH_YOUR_SSID'
password = 'REPLACE_WITH_YOUR_PASSWORD'
```

Insert your unique API key from the Webhooks IFTTT service:

```
api_key = 'REPLACE_WITH_YOUR_WEBHOOKS_IFTTT_API_KEY'
```

The following lines connect the ESP32/ESP8266 to your router, and print the board IP address:

```
station = network.WLAN(network.STA_IF)

station.active(True)
station.connect(ssid, password)

while station.isconnected() == False:
  pass
print('Connection successful')
print(station.ifconfig())
```

Then, you create an I2C instance called `i2c`. You need this to create an I2C connection with the BME280 sensor. You need to pass as argument the SCL and SDA pins, as well as a frequency. Don't worry about the frequency, a value of 10000 Hz works just fine for this sensor.

The ESP32 default I2C pins are GPIO 22 (SCL) and GPIO 21 (SDA). The ESP8266 default I2C pins are GPIO 5 (SLC) and GPIO 4 (SDA).

Use the following line if you're using an ESP32 board:

```
# ESP32 - Pin assignment
i2c = I2C(scl=Pin(22), sda=Pin(21), freq=10000)
```

Comment the previous line and uncomment the following if you're using an ESP8266 board:

```
# ESP8266 - Pin assignment
#i2c = I2C(scl=Pin(5), sda=Pin(4), freq=10000)
```

Then, use `try` and `except` statements. In the `try` statement, we read from the sensor and publish the sensor readings. If one of these tasks fail, the `except` block runs and prints the 'Failed to read/publish sensor readings.' message.

```
except OSError as e:
  print('Failed to read/publish sensor readings.')
```

To take sensor readings, first create an instance of the BME280 object as follows:

```
bme = BME280.BME280(i2c=i2c)
```

After that, to get temperature, humidity and pressure, we just need to do as follows:

```
temp = bme.temperature
hum = bme.humidity
pres = bme.pressure
```

Save the temperature on the `temp` variable, the humidity on the `hum` variable, and the pressure on the `pres` variable.

To send the readings to the IFTTT services, we need to make an HTTP POST request. A general POST request in MicroPython using the `urequests` library has the following format:

```
request= requests.post(url, json=json_data, headers=headers)
```

The first parameter is the URL in which you will make the request. As we've seen in the IFTTT Applet, we need to use the following URL (in which `api_key` should be replaced with your own API key):

```
http://maker.ifttt.com/trigger/bme280/with/key/api_key
```

The JSON parameter should contain data in JSON format.

JSON stands for JavaScript Object Notation (JSON), and it is a standard for exchanging data. In JSON syntax, data is represented in name/value pairs. Each name is followed by a colon (:), and the value pairs are separated with comas.

The `sensor_readings` variable holds the temperature, humidity, and pressure values in JSON format.

```
sensor_readings = {'value1':temp, 'value2':hum, 'value3':pres}
```

The `headers` parameter contains information about the request. For our request, the headers should be as follows:

```
request_headers = {'Content-Type': 'application/json'}
```

Make the request using the `post()` method with the data we've defined earlier:

```
request = urequests.post(
    'http://maker.ifttt.com/trigger/bme280/with/key/' + api_key,
    json=sensor_readings,
    headers=request_headers)
```

Finally, print the request for debugging purposes and close the request:

```
print(request.text)
request.close()
```

Demonstration

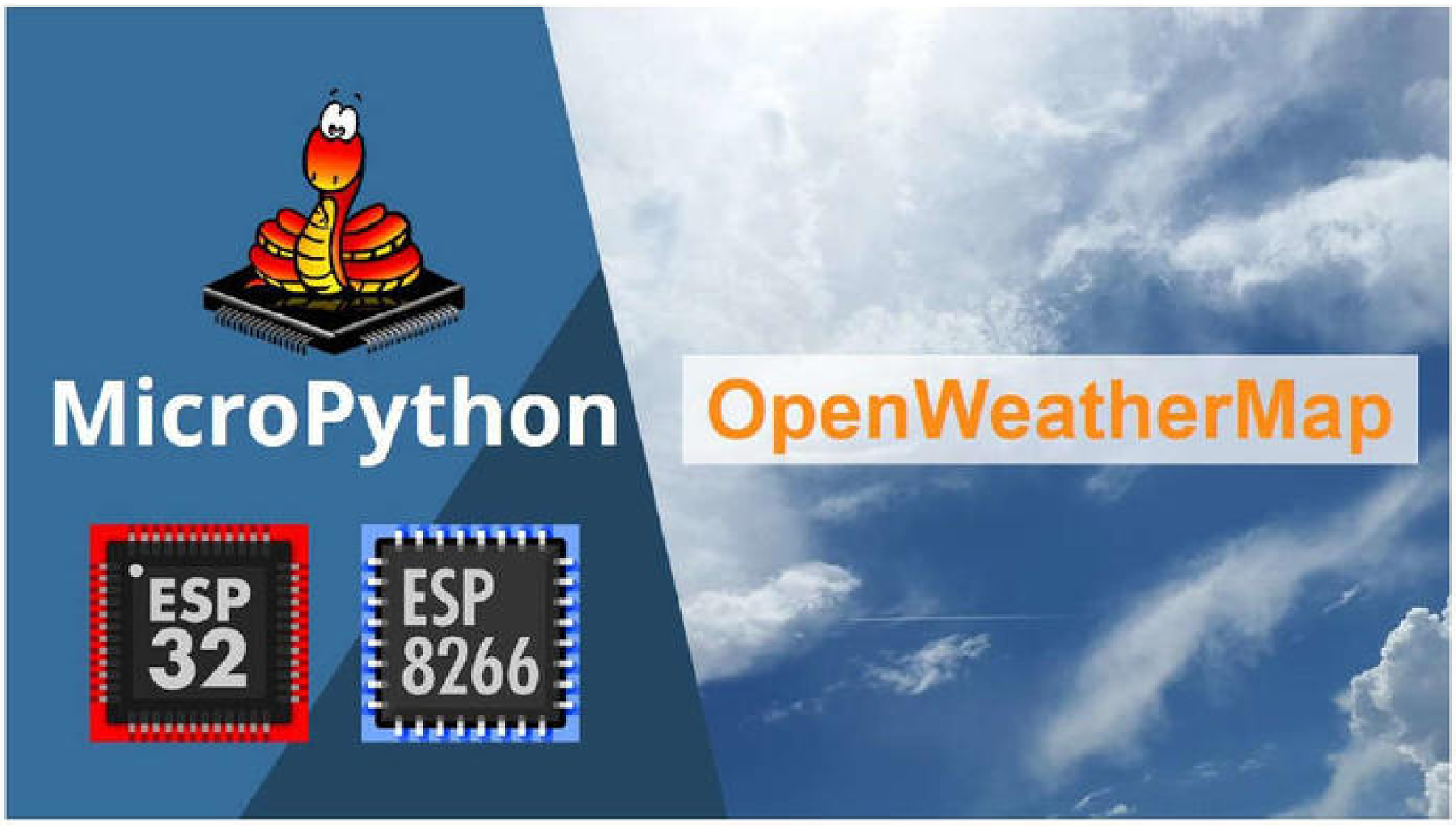
After uploading the *boot.py* file to your ESP32 or ESP8266 board, you should receive an email with the latest sensor readings.

Note: if you don't get sensor readings, reset the ESP32/ESP8266 by pressing the EN/RST on-board button.

On the Shell, you should get a similar message:

```
Connection successful
('192.168.1.148', '255.255.255.0', '192.168.1.254', '192.168.1.254')
{'value3': '1009.71hPa', 'value1': '22.60C', 'value2': '61.02%'}
Congratulations! You've fired the bme280 event
```

Getting Data Using an API



In this Unit you'll learn how to make API requests to access data. As an example, we'll use the OpenWeatherMap API. This API is free and provides lots of useful information about the weather in almost any location in the world.

Using the OpenWeatherMap API

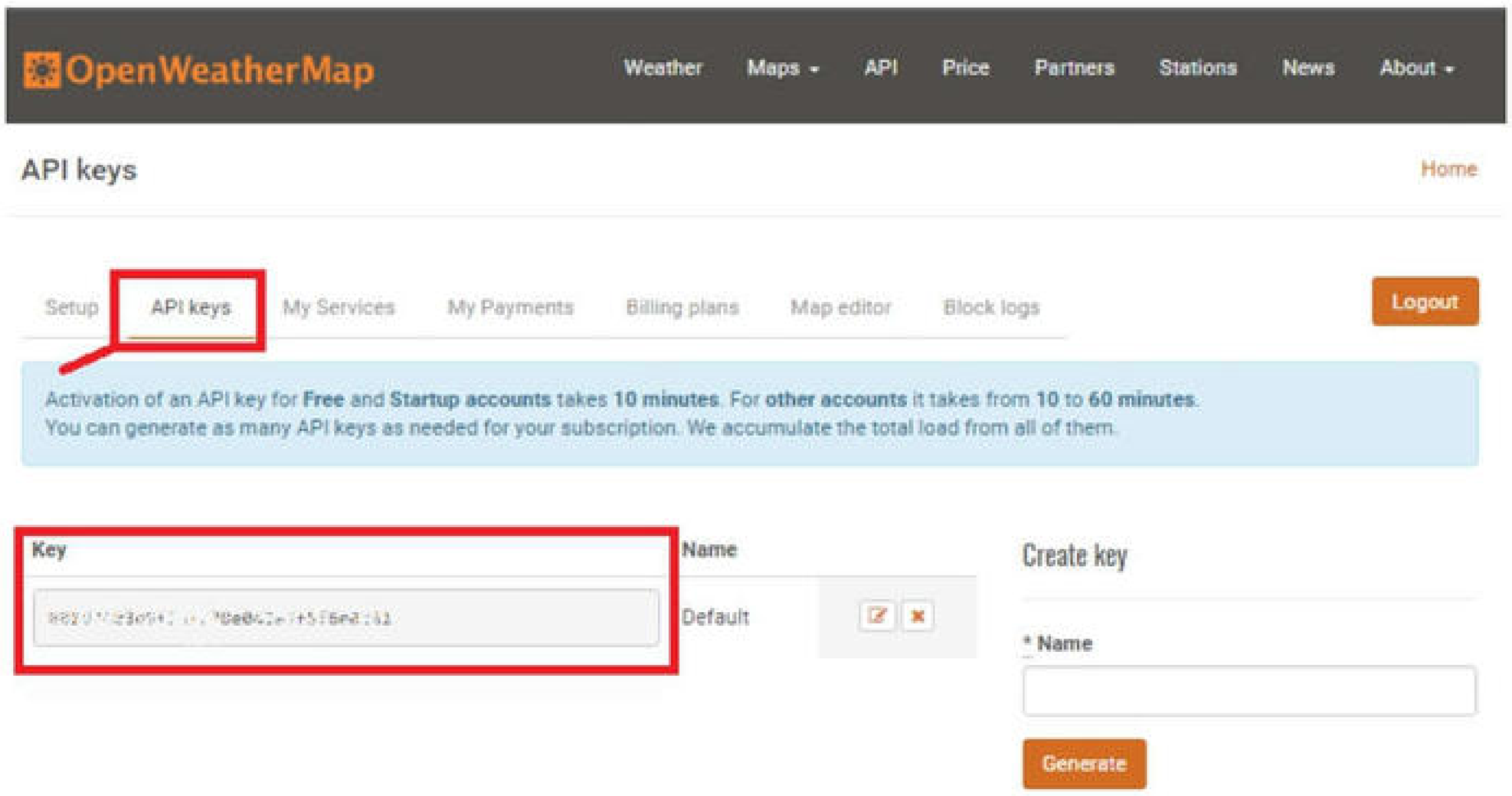
An application programming interface (API) is a set of functions written by software developers to enable anyone to use their data or services. For example, the **OpenWeatherMap** project (https:// openweathermap.org/) has an API that enables users to request weather data using many different programming languages.

OpenWeatherMap

In this project, you'll use that API to request the day's weather forecast for your chosen location. Learning to use APIs is a great skill because it allows you access to a wide variety of constantly changing information, such as current stock prices, currency exchange rates, the latest news, traffic updates, tweets, and much more.

OpenWeatherMap's free plan provides everything you need to complete this project. To use the API you need an API key, known as the APIID. To get the APIID:

1. Open a browser and go to https://openweathermap.org/appid/
2. Press the **Sign up** button and create a free account.
3. You'll be presented with a dashboard that contains several tabs. Select the **API keys** tab, as shown in the following figure.



4. On the API keys tab, you'll see a default key (highlighted in a red rectangle in figure above); this is a unique key you'll need to pull information from the site. Copy and paste this key somewhere; you'll need it in a moment.
5. To pull information on weather in your chosen location, enter the following URL:

```
http://api.openweathermap.org/data/2.5/weather?q=your_city,
your_country_code&APPID=your_unique_API_key
```

Replace **your_city** with the city you want data for, **your_country_code** with the country code for that city, and **your_unique_API_key** with the unique API key from step 4. For example, the updated API URL for the city of Porto, Portugal, would be:

```
http://api.openweathermap.org/data/2.5/weather?q=Porto,
PT&APPID=801d2603e9f2e1c70e042e4f5f6e0---
```

Note: API keys are unique to the user and shouldn't be shared with anyone.

6. Copy your URL into your browser, and the API will return a bunch of information corresponding to your local weather. We got the following information about the weather in Porto, Portugal, on the day we wrote this project.


```
{"coord":{"lon":8.61,"lat":41.15},"weather":[{"id":803,"main":
"Clouds","description":"broken  clouds","icon":"04d"}],"base":"
stations","main":{"temp":285.67,"pressure":1017,"humidity":71,
"temp_min":285.15,"temp_max":286.15},"visibility":10000,"wind"
:{"speed":5.7,"deg":340},"clouds":{"all":75},"dt":1548781200,"
sys":{"type":1,"id":6900,"message":0.0036,"country":"PT","sunr
ise":1548748072,"sunset":1548784072},"id":2735943,"name":"Port
o","cod":200}
```


Next, you'll see how to organize this data with tabs and paragraphs to make it more readable.

Understanding JSON Syntax

As you can see, the weather data for your chosen location is stored in a specific way, with symbols like **{} [] : ""** and **,**. This syntax is JavaScript Object Notation (JSON), a standard for exchanging data in a convenient way. In JSON syntax:

- Data is represented in name/value pairs.
- Each name is followed by a colon (:).
- Name/value pairs are separated with commas.
- Curly brackets hold objects.
- Square brackets hold arrays.

Below is the same data returned earlier but with spaces and indentation in a way that is easier to understand.

```
{
   "coord":{
      "lon":-8.61,
      "lat":41.15
   },
   "weather":[{
         "id":803,
```

```
            "main":"Clouds",
            "description":"broken clouds",
            "icon":"04d"
        }
    ],
    "base":"stations",
    "main":{
        "temp":285.67,
        "pressure":1017,
        "humidity":71,
        "temp_min":285.15,
        "temp_max":286.15
    },
    "visibility":10000,
    "wind":{
        "speed":5.7,
        "deg":340
    },
    "clouds":{
        "all":75
    },
    "dt":1548781200,
    "sys":{
        "type":1,
        "id":6900,
        "message":0.0036,
        "country":"PT",
        "sunrise":1548748072,
        "sunset":1548784072
    },
    "id":2735943,
    "name":"Porto",
    "cod":200
}
```


Now you can easily see all the useful information that the OpenWeatherMap API provides.

Making an API Request

After having a URL that returns your local weather data, we'll write a MicroPython script that returns temperature, humidity, pressure and wind for your chosen location. This information is then printed on the Shell.

boot.py

Copy the following script to the *boot.py* file. The goal of this project is to demonstrate how to decode JSON data from an API, so you can apply this concept to your own projects.

```
#import necessary libraries
import time

try:
  import urequests as requests
except:
  import requests

try:
  import ujson as json
except:
  import json

import network

import esp
esp.osdebug(None)

import gc
gc.collect()

ssid = 'REPLACE_WITH_YOUR_SSID'
password = 'REPLACE_WITH_YOUR_PASSWORD'

city = 'REPLACE_WITH_YOUR_CITY_NAME'
country_code = 'REPLACE_WITH_YOUR_2_LETTER_COUNTRY_CODE'

open_weather_map_api_key = ' REPLACE_WITH_YOUR_API_KEY '

station = network.WLAN(network.STA_IF)
```

```
station.active(True)
station.connect(ssid, password)

while station.isconnected() == False:
  pass

print('Connection successful')
print(station.ifconfig())

#set your unique OpenWeatherMap.org URL
open_weather_map_url =
'http://api.openweathermap.org/data/2.5/weather?q=' + city +
',' + country_code + '&APPID=' + open_weather_map_api_key

weather_data = requests.get(open_weather_map_url)
print(weather_data.json())

# Location (City and Country code)
location = 'Location: ' + weather_data.json().get('name') + '
- ' + weather_data.json().get('sys').get('country')
print(location)

# Weather Description
description = 'Description: ' +
weather_data.json().get('weather')[0].get('main')
print(description)

# Temperature
raw_temperature = weather_data.json().get('main').get('temp')-273.15

# Temperature in Celsius
temperature = 'Temperature: ' + str(raw_temperature) + '*C'
#uncomment for temperature in Fahrenheit
#temperature = 'Temperature: ' +
str(raw_temperature*(9/5.0)+32) + '*F'
print(temperature)

# Pressure
pressure = 'Pressure: ' +
str(weather_data.json().get('main').get('pressure')) + 'hPa'
print(pressure)

# Humidity
humidity = 'Humidity: ' +
str(weather_data.json().get('main').get('humidity')) + '%'
print(humidity)
```

```
# Wind
wind = 'Wind: ' +
str(weather_data.json().get('wind').get('speed')) + 'mps ' +
str(weather_data.json().get('wind').get('deg')) + '*'
print(wind)
```

SOURCE CODE
https://github.com/RuiSantosdotme/ESP-MicroPython/blob/master/code/WiFi/HTTP_API_OpenWeatherMap/boot.py

Let's take a quick look on how the code works.

Importing libraries

You start by importing the necessary libraries:

```
import time

try:
  import urequests as requests
except:
  import requests

try:
  import ujson as json
except:
  import json

import network
```

You need the `urequests` and `ujson` library to make API requests and access the data. You need the `network` library to connect the ESP32/ESP8266 board to your router via Wi-Fi.

Entering your credentials

Then, you need to enter your network credentials:

```
ssid = 'REPLACE_WITH_YOUR_SSID'
password = 'REPLACE_WITH_YOUR_PASSWORD'
```

Enter the city you want to get data for, as well as the country code in the following variables:

```
city = 'REPLACE_WITH_YOUR_CITY_NAME'
country_code = 'REPLACE_WITH_YOUR_2_LETTER_COUNTRY_CODE'
```

You also need to enter your unique API key on the following line:

```
open_weather_map_api_key = ' REPLACE_WITH_YOUR_API_KEY '
```

Connecting to the network

Then, we make the usual procedures to connect the ESP32/ESP8266 to your local network.

```
station = network.WLAN(network.STA_IF)

station.active(True)
station.connect(ssid, password)

while station.isconnected() == False:
  pass

print('Connection successful')
print(station.ifconfig())
```

Making the API request

Then, we create a variable called `open_weather_map_url` to hold the API URL with the credentials you've entered earlier: city, country code, and API key.

```
open_weather_map_url =
'http://api.openweathermap.org/data/2.5/weather?q=' + city +
',' + country_code + '&APPID=' + open_weather_map_api_key
```

Finally, you make the API request as follows:

```
weather_data = requests.get(open_weather_map_url)
```

Basically, you create a variable called `weather_data` in which you store the data returned after the API request. To make an API request for information, you use the command `requests.get(your_url)`, with your URL as the argument.

Then, print the raw data on the Shell. To get the data, you first need to convert the `weather_data` variable to JSON with the `.json()` method.

```
print(weather_data.json())
```

After that, we have several lines of code that work in a similar way. These lines access specific pieces of information on the API response. For example, as seen in the "Understanding JSON syntax" section, inside the `'wind'`, you have `'speed'` and `'deg'`.

```
"wind":{
    "speed":5.7,
      "deg":340
```

To access speed:

```
weather_data.json().get('wind').get('speed')
```

Basically, you need to use the `get()` method to access specific data.

The same way, to access the location, you use:

```
weather_data.json().get('sys').get('country')
```

Demonstration

Copy the previous script to the ESP32/ESP8266 *boot.py* file. After uploading the code, press the EN/RESET button. It should print the weather data for your specific location.

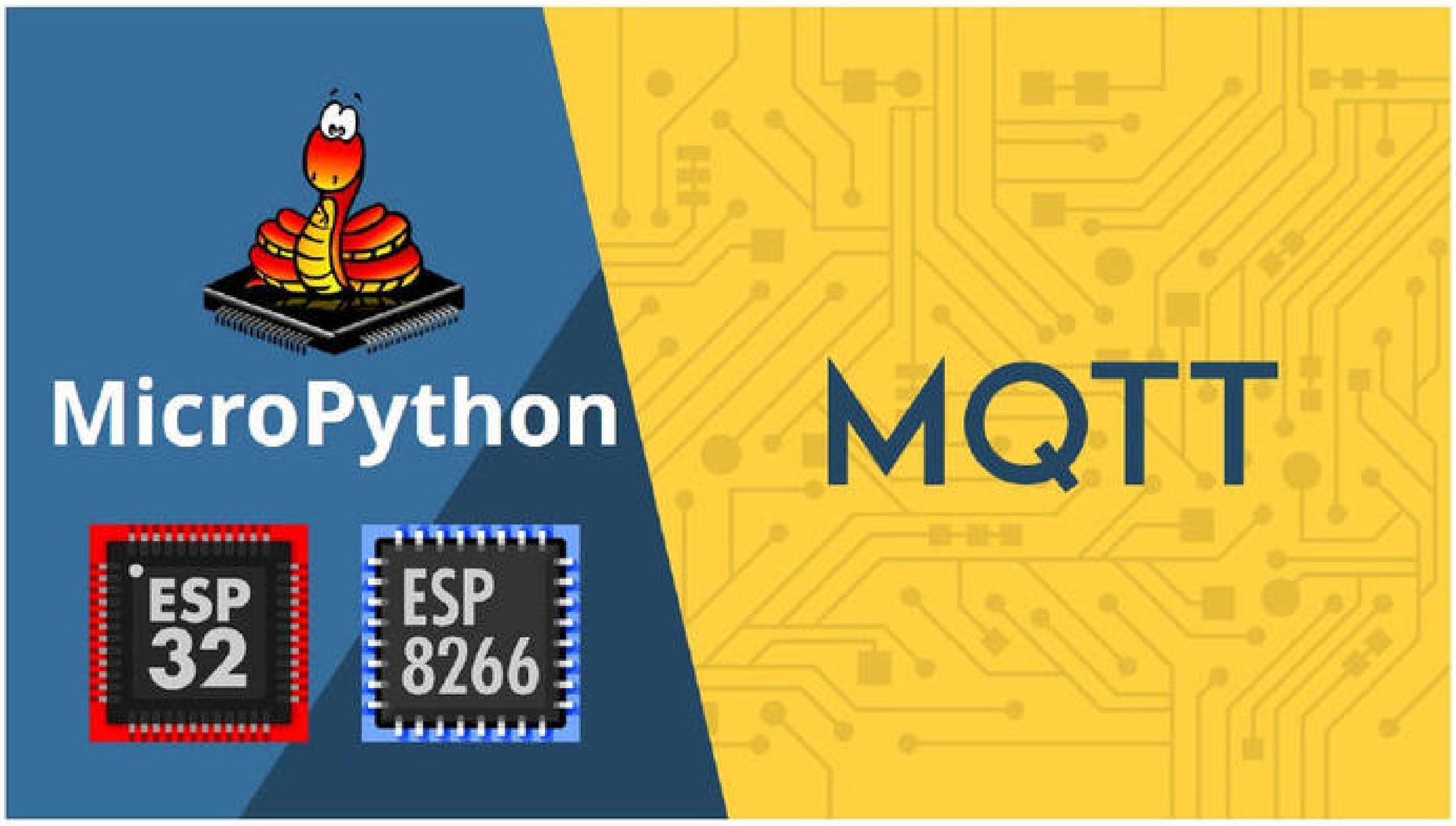
```
Connection successful
('192.168.1.77', '255.255.255.0', '192.168.1.254', '192.168.1.254')
{'sys': {'message': 0.0033, 'country': 'PT', 'sunrise': 1548748049, 'sunset': 1548784058, 'id': 6900,
'type': 1}, 'name': 'Gondomar', 'main': {'pressure': 1017, 'humidity': 76, 'temp_min': 284.15,
'temp_max': 285.15, 'temp': 284.66}, 'cod': 200, 'base': 'stations', 'wind': {'speed': 3.6, 'deg':
340}, 'id': 2738925, 'clouds': {'all': 75}, 'coord': {'lon': -8.53, 'lat': 41.14}, 'dt': 1548784800,
'weather': [{'id': 803, 'icon': '04n', 'main': 'Clouds', 'description': 'broken clouds'}],
'visibility': 10000}
Location: Gondomar - PT
Description: Clouds
Temperature: 11.51*C
Pressure: 1017hPa
Humidity: 76%
Wind: 3.6mps 340*
```

You can also try to request data on other locations by changing the `city` and `country_code` variables.

MODULE 5

MQTT Protocol

Introducing MQTT



This Unit is an introduction to MQTT and how to use it with the ESP32.

MQTT stands for Message Queuing Telemetry Transport. It is a lightweight publish and subscribe system where you can publish and receive messages as a client.

MQTT is a simple messaging protocol, designed for constrained devices with low-bandwidth. It's the perfect solution for Internet of Things applications. MQTT allows you to send commands to control outputs, read and publish data from sensor nodes and much more.

MQTT Basic Concepts

In MQTT there are a few basic concepts that you need to understand:

- Publish/Subscribe
- Messages
- Topics
- Broker

Publish/subscribe

The first concept is the ***publish*** *and* ***subscribe*** system. In a publish and subscribe system, a device can publish a message on a topic, or it can be subscribed to a topic to receive messages.

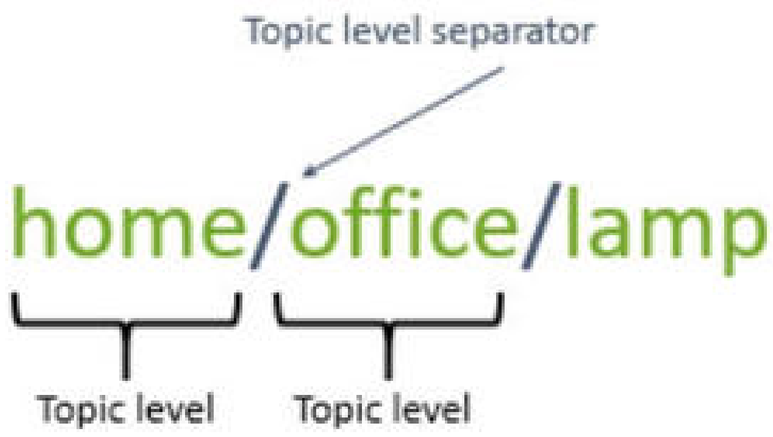
- For example, Device 1 publishes on a topic;
- Device 2 is subscribed to the same topic that Device 1 is publishing in;
- So, Device 2 receives the message.

Messages are pieces of information exchanged between your devices: whether it's a command or data.

Topics

Another important concept are the ***topics***. Topics are the way you register interest for incoming messages or how you specify where you want to publish the messages.

Topics are represented with strings separated by a forward slash. Each forward slash indicates the topic level. Here's an example on how you would create a topic for a lamp in your home office:



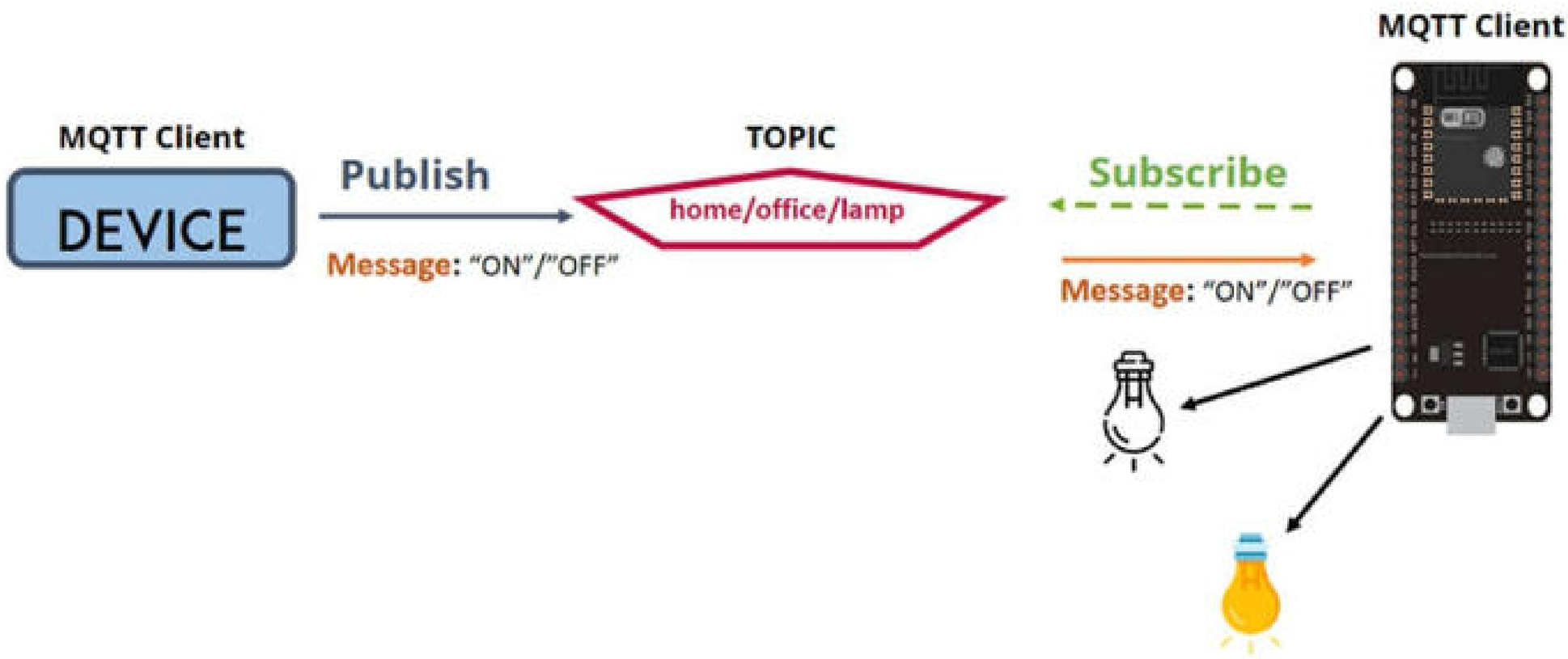
Note: topics are case-sensitive, which makes the following topics different.

home/office/lamp

≠

Home/Office/Lamp

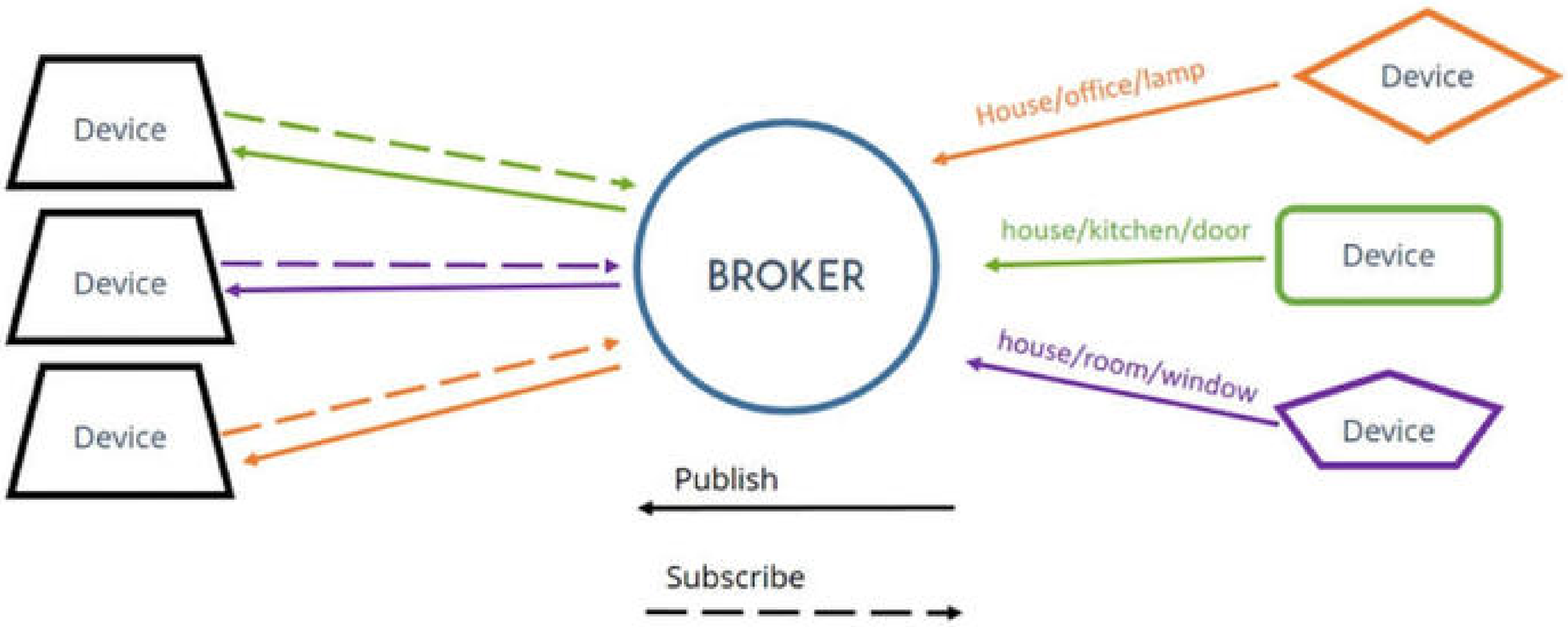
If you would like to turn on a lamp in your home office using MQTT and the ESP32 you can imagine the following scenario:



- You have a device that publishes "on" and "off" messages on the **home/office/lamp** topic.
- The ESP32 that controls your lamp, is subscribed to that topic.
- Therefore, when a new message is published on that topic, the ESP32 receives the "on" or "off" message and turns the lamp on or off.
- This first device can be an ESP32, an ESP8266, or a Home Automation controller platform like Node-RED, Home Assistant, Domoticz, or OpenHAB, for example.

Broker

Finally, you also need to be familiar with the term ***broker***. The broker is primarily responsible for receiving all messages, filtering the messages, decide who is interested in them and then send the messages to all subscribed clients.



You can use several brokers. For example, you can use the **Mosquitto** broker hosted on a Raspberry Pi, or you can use a cloud MQTT broker.

We're going to use the Mosquitto broker installed in a Raspberry Pi.

Wrapping Up

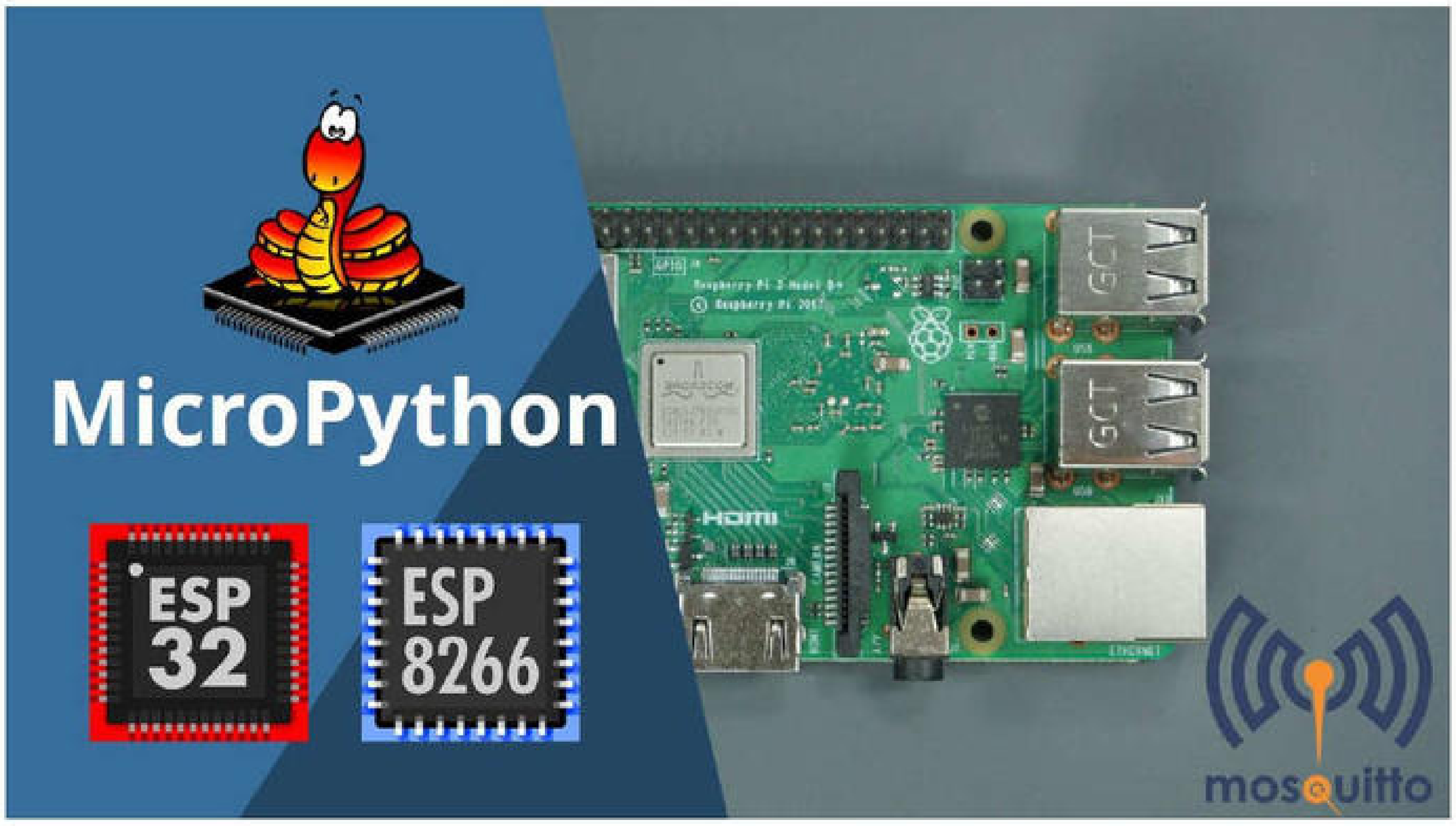
In summary:

- MQTT is a communication protocol very useful in Internet of Things projects;
- In MQTT, devices can publish messages on specific topics and can be subscribed to other topics to receive messages;
- You need a broker when using MQTT. It receives all the messages and sends them to the subscribed devices.

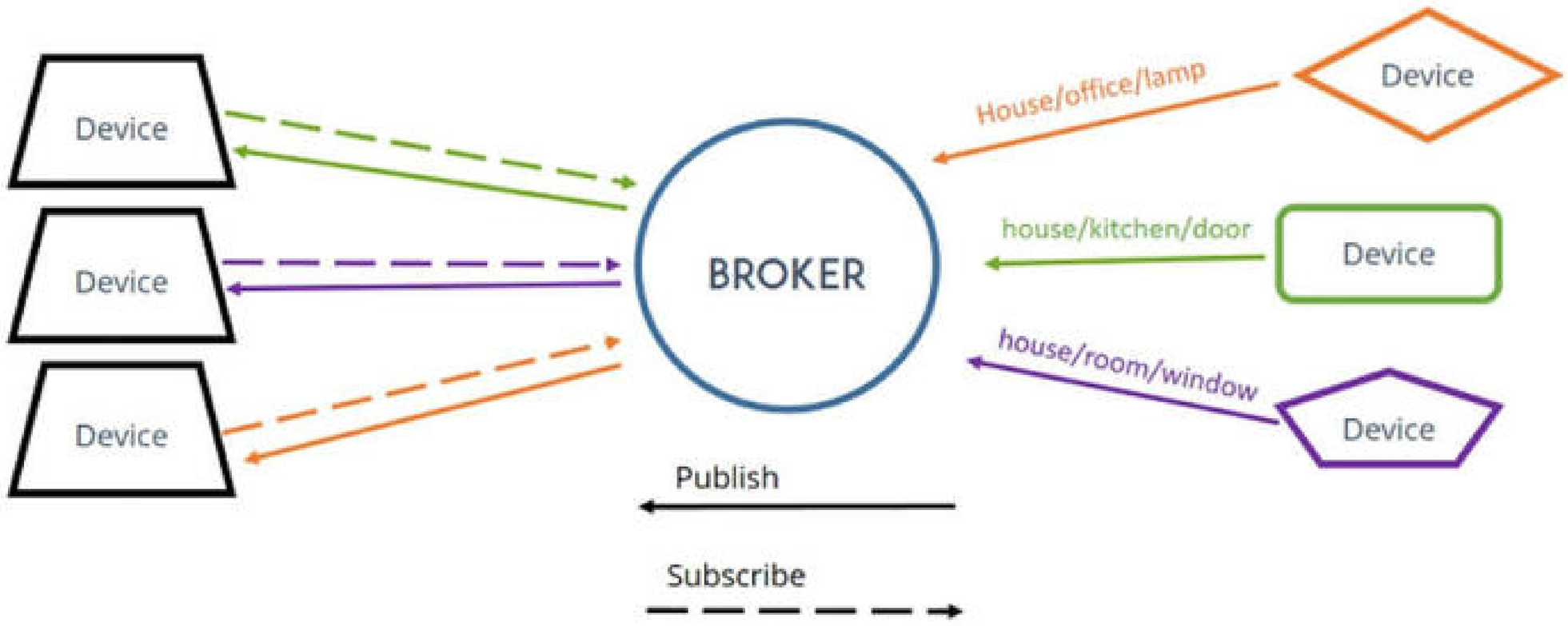
In the next Units, you'll learn how to:

- Install an MQTT broker;
- Exchange data between two ESP32/ESP8266 boards;
- Exchange data between ESP32/ESP8266 and Node-RED.

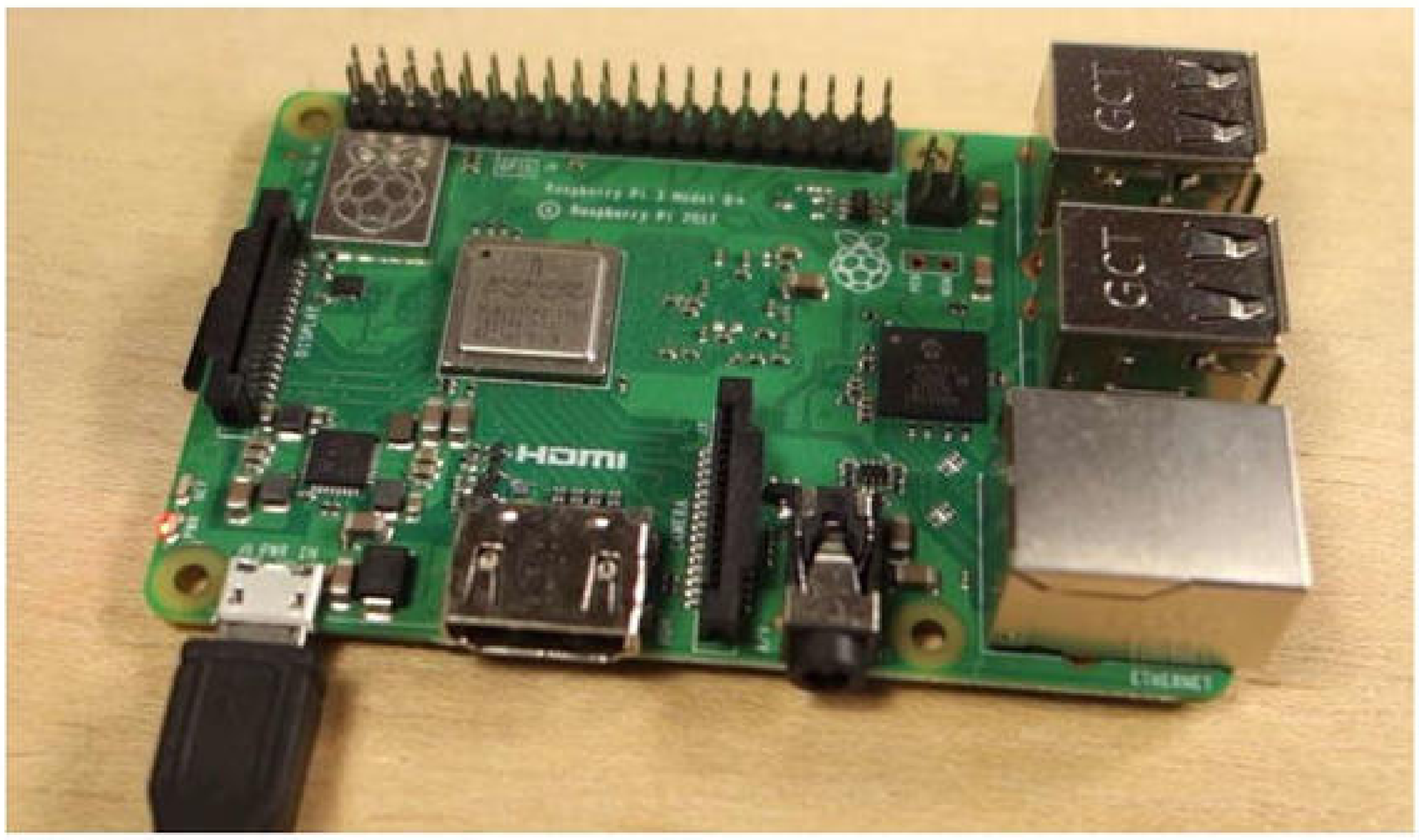
Installing Mosquitto MQTT Broker on a Raspberry Pi



In this Unit, you're going to install the Mosquitto Broker on a Raspberry Pi. The broker is primarily responsible for receiving all messages, filtering the messages, decide who is interested in them and then, publishing the message to all subscribed clients.



You can use several brokers. Throughout this eBook, we're going to use the Mosquitto Broker installed on a Raspberry Pi.



Note: you can also use a free Cloud MQTT broker (for a maximum of 5 connected devices).

Prerequisites

Before continuing with this tutorial:

- You should be familiar with the Raspberry Pi board – read Getting Started with Raspberry Pi;
- You should have the Raspbian or Raspbian Lite operating system installed in your Raspberry Pi – read Installing Raspbian Lite, Enabling and Connecting with SSH;
- You also need the following hardware:
  - Raspberry Pi board – read Best Raspberry Pi Starter Kits
  - MicroSD Card – 16GB Class10
  - Raspberry Pi Power Supply (5V 2.5A)

After having your Raspberry Pi board prepared with Raspbian OS, you can continue with this Unit. Let's install the Mosquitto Broker.

Installing Mosquitto Broker on Raspbian OS

Open a new Raspberry Pi terminal window:

To install the Mosquitto Broker enter these next commands:

```
pi@raspberry:~ $ sudo apt update
pi@raspberry:~ $ sudo apt install -y mosquitto mosquitto-clients
```

To make Mosquitto auto start on boot up (this means Mosquitto will start automatically when you power your Raspberry Pi) enter:

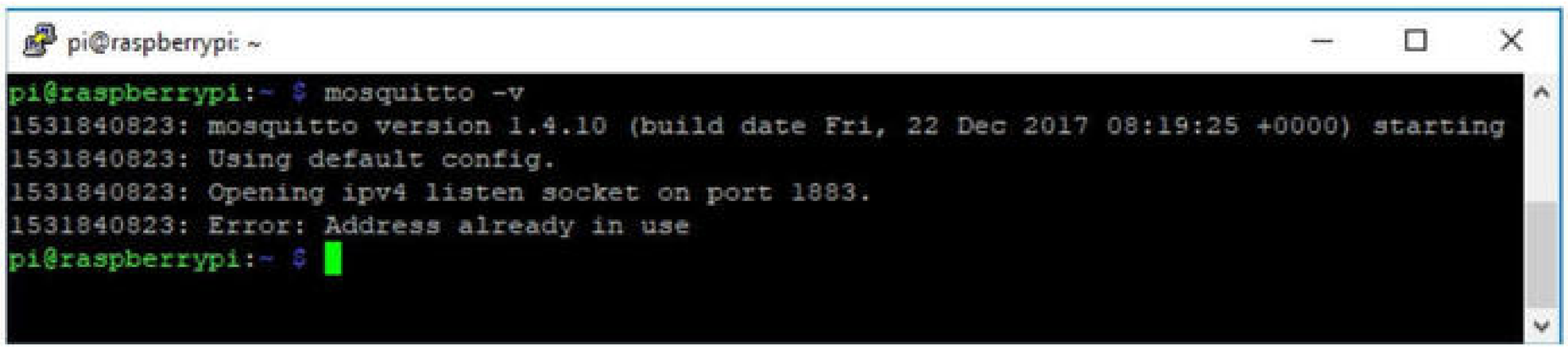
```
pi@raspberry:~ $ sudo systemctl enable mosquitto.service
```

Testing Installation

Send the command:

```
pi@raspberry:~ $ mosquitto -v
```

This returns the Mosquitto version that is currently running in your Raspberry Pi. It should be 1.4.X or above.

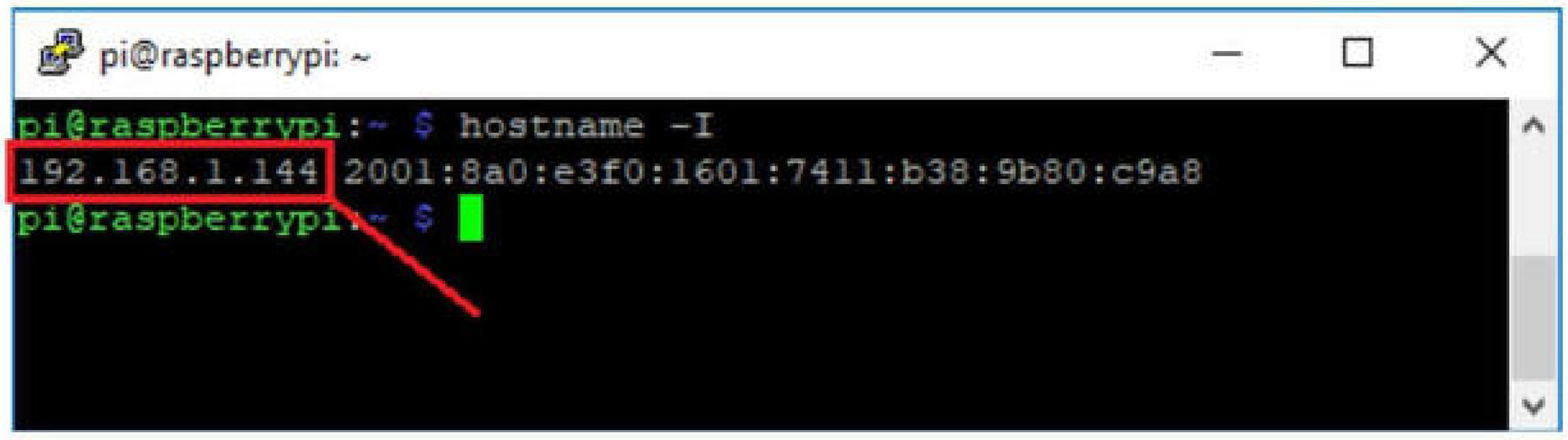


Note: sometimes the command `mosquitto -v` prompts a warning message saying "`Error: Address already in use`". That warning message means that your Mosquitto Broker is already running, so don't worry about that.

Raspberry Pi IP Address

To retrieve your Raspberry Pi IP address, type the next command in your Terminal window:

```
pi@raspberry:~ $ hostname -I
```



In our case, the Raspberry Pi IP address is **192.168.1.144**. Save your IP address. You'll need it in the next Units to connect the ESP32/ESP8266 with the broker.

Wrapping Up

These were the steps to install the Mosquitto broker on a Raspberry Pi. In the next Unit, we'll set up two ESP32 boards as MQTT clients and you'll see how everything ties together with practical examples.

MQTT – Establishing a Two-way Communication

In this Unit, we're going to demonstrate how to use MQTT to exchange data between two ESP32/ESP8266 boards. As an example, we'll exchange simple text messages between them. The idea is to use the concepts learned here to exchange sensor readings, or commands.

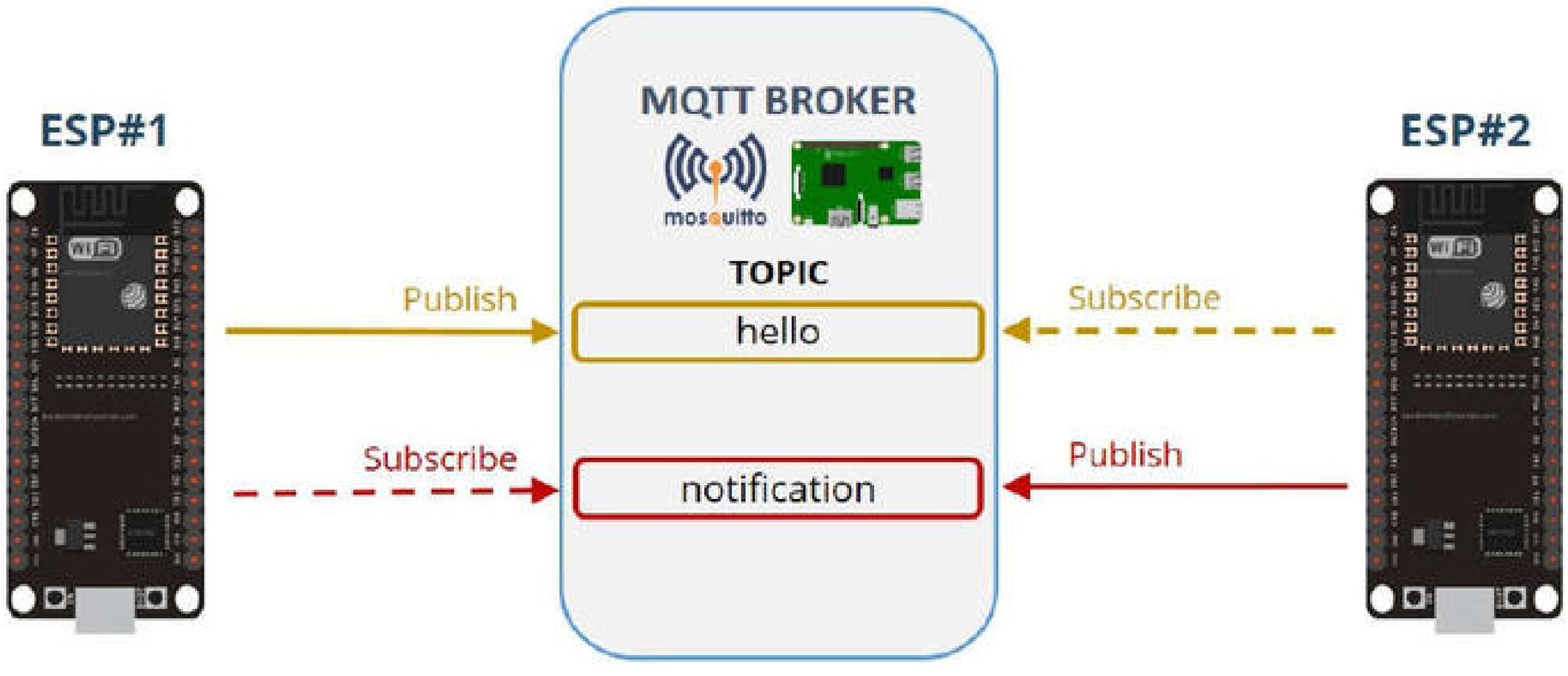
For this project, you need a Mosquitto MQTT broker running. We're using a Raspberry Pi to set up the Mosquitto broker (follow the instructions in the previous Unit).

Project Overview

Here's a high-level overview of the project we'll build:

- ESP#1 publishes messages on the ***hello*** topic. It publishes a "Hello" message followed by a counter (Hello 1, Hello 2, Hello 3, ...). It publishes a new message every 5 seconds.
- ESP#1 is subscribed to the ***notification*** topic to receive notifications from the ESP#2 board.

- ESP#2 is subscribed to the ***hello*** topic. ESP #1 is publishing in this topic. Therefore, ESP#2 receives ESP#1 messages.
- When ESP#2 receives the messages, it sends a message saying 'received'. This message is published on the ***notification*** topic. ESP#1 is subscribed to that topic, so it receives the message.



Preparing ESP#1

Let's start by preparing ESP#1:

- It is subscribed to the ***notification*** topic
- It publishes on the ***hello*** topic

Importing umqtttsimple

To use MQTT with the ESP32/ESP8266 and MicroPython, you need to install the `umqttsimple` library.

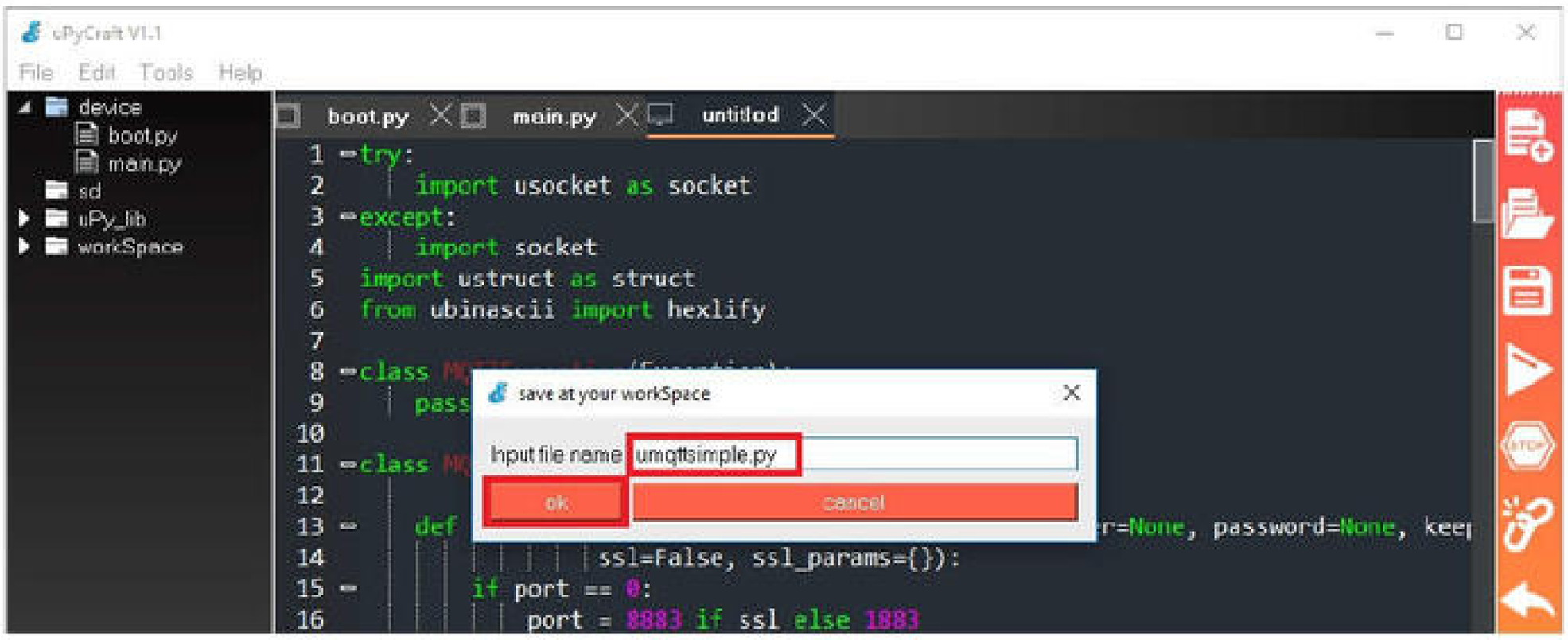
Create a new file by pressing the **New File** button.

Copy the `umqttsimple` library code into it. You can access the `umqttsimple` library code in the following link:

https://github.com/RuiSantosdotme/ESP-MicroPython/blob/master/code/MQTT/umqttsimple.py

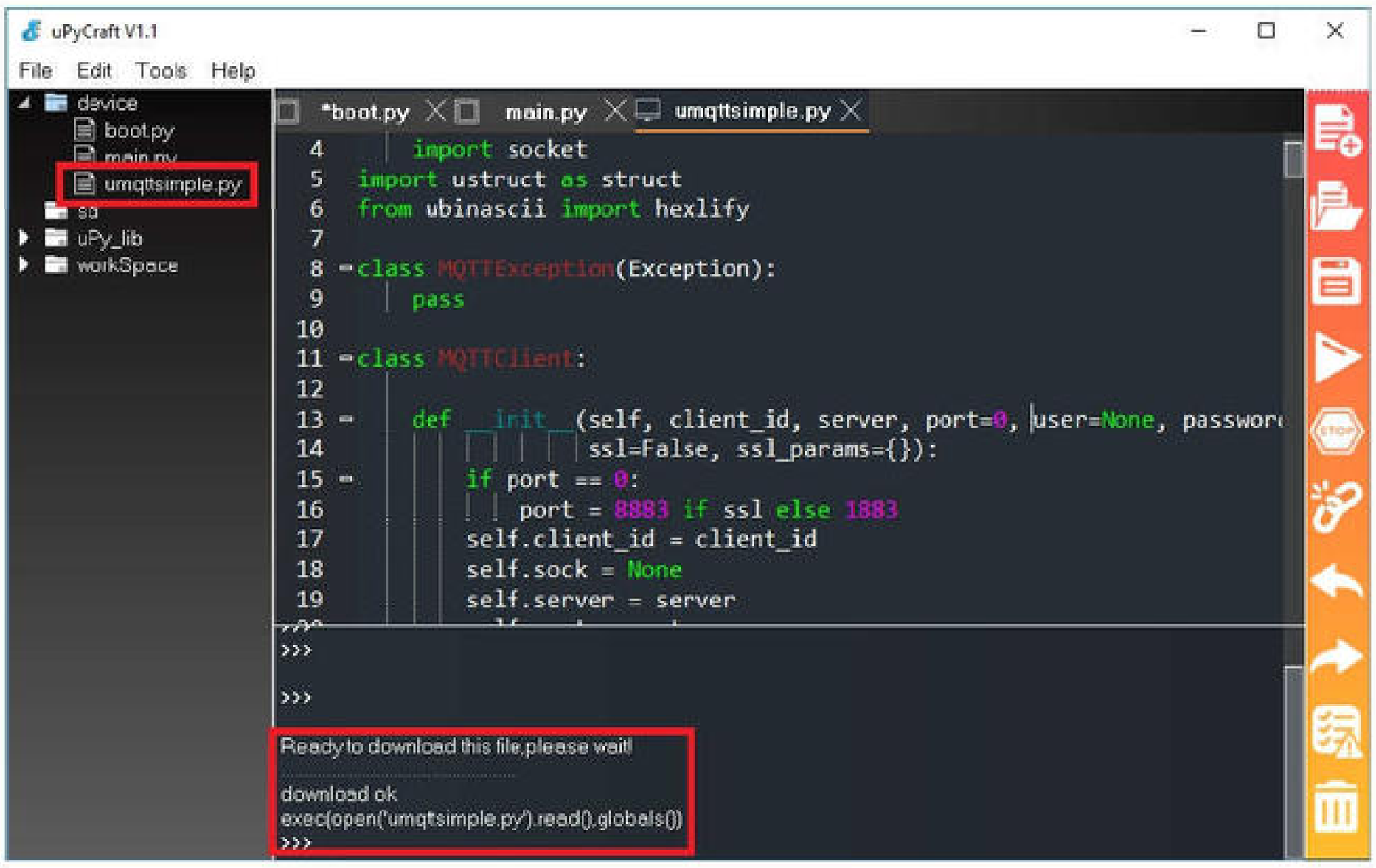
Save the file by pressing the **Save** button.

Call this new file "**umqttsimple.py**" and press **ok**.



Click the **Download and Run** button.

The file should be saved on the *device* folder with the name "**umqttsimple.py**" as highlighted in the figure below.



Now, you can use the library functionalities in your code by importing the library.

Note: if you're using Thonny IDE, you just need to:

1. Copy the library code to a new file.

2. Name that file *umqttsimple.py*.

3. Go to **Device ▸ Upload current script with the current name**.

boot.py

Open the *boot.py* file and copy the following code to ESP#1.

```
import time
from umqttsimple import MQTTClient
import ubinascii
import machine
import micropython
import network
import esp
esp.osdebug(None)
import gc
gc.collect()

ssid = 'REPLACE_WITH_YOUR_SSID'
password = 'REPLACE_WITH_YOUR_PASSWORD'
mqtt_server = 'REPLACE_WITH_YOUR_MQTT_BROKER_IP'
#EXAMPLE IP ADDRESS
#mqtt_server = '192.168.1.144'
client_id = ubinascii.hexlify(machine.unique_id())
topic_sub = b'notification'
topic_pub = b'hello'
last_message = 0
message_interval = 5
counter = 0

station = network.WLAN(network.STA_IF)
station.active(True)
station.connect(ssid, password)
while station.isconnected() == False:
  pass
print('Connection successful')
print(station.ifconfig())
```

SOURCE CODE

https://github.com/RuiSantosdotme/ESP-MicroPython/blob/master/code/MQTT/MQTT_Hello_World/ESP_1/boot.py

How the Code Works

You need to import all the following libraries:

```
import time
from umqttsimple import MQTTClient
import ubinascii
import machine
import micropython
import network
import esp
```

Set the debug to `None` and activate the garbage collector.

```
esp.osdebug(None)
import gc
gc.collect()
```

In the following variables, you need to enter your network credentials and your broker IP address.

```
ssid = 'REPLACE_WITH_YOUR_SSID'
password = 'REPLACE_WITH_YOUR_PASSWORD'
mqtt_server = 'REPLACE_WITH_YOUR_MQTT_BROKER_IP'
```

For example, our broker IP address is: 192.168.1.144.

Note: check the previous Unit to see how to get your MQTT broker IP address.

To create an MQTT client, we need to get the ESP unique ID. That's what we do in the following line (it is saved on the `client_id` variable).

```
client_id = ubinascii.hexlify(machine.unique_id())
```

Next, write the topic the ESP#1 is subscribed to, and the topic it will be publishing messages:

```
topic_sub = b'notification'
topic_pub = b'hello'
```

Then, create the following variables:

```
last_message = 0
message_interval = 5
counter = 0
```

The `last_message` variable holds the last time a message was sent. The `message_interval` is the time between each message sent. Here, we're setting it to 5 seconds (this means a new message will be sent every 5 seconds). The `counter` variable is simply a counter to be added to the message.

After that, we make the usual procedures to connect to the network.

```
station = network.WLAN(network.STA_IF)

station.active(True)
station.connect(ssid, password)

while station.isconnected() == False:
  pass

print('Connection successful')
print(station.ifconfig())
```

main.py

In the *main.py* file is where we'll write the code to publish and receive the messages. Copy the following code to your *main.py* file.

```
def sub_cb(topic, msg):
  print((topic, msg))
  if topic == b'notification' and msg == b'received':
    print('ESP received hello message')

def connect_and_subscribe():
  global client_id, mqtt_server, topic_sub
  client = MQTTClient(client_id, mqtt_server)
  client.set_callback(sub_cb)
  client.connect()
  client.subscribe(topic_sub)
  print('Connected to %s MQTT broker, subscribed to %s topic' %
(mqtt_server, topic_sub))
  return client

def restart_and_reconnect():
  print('Failed to connect to MQTT broker. Reconnecting...')
  time.sleep(10)
  machine.reset()

try:
```

```
  client = connect_and_subscribe()
except OSError as e:
  restart_and_reconnect()

while True:
  try:
    client.check_msg()
    if (time.time() - last_message) > message_interval:
      msg = b'Hello #%d' % counter
      client.publish(topic_pub, msg)
      last_message = time.time()
      counter += 1
  except OSError as e:
    restart_and_reconnect()
```

SOURCE CODE
https://github.com/RuiSantosdotme/ESP-MicroPython/blob/master/code/MQTT/MQTT_Hello_World/ESP_1/main.py

How the Code Works

The first thing you should do is creating a callback function that will run whenever a message is published on a topic the ESP is subscribed to.

Callback function

The function should accept as parameters the topic and the message.

```
def sub_cb(topic, msg):
  print((topic, msg))
  if topic == b'notification' and msg == b'received':
    print('ESP received hello message')
```

In our callback function, start by printing the topic and the message. Then, check if the message was published on the ***notification*** topic, and if the content of the message is 'received'. If this if statement is True, it means that ESP#2 received the 'hello' message sent by ESP#1.

Basically, this callback function handles what happens when a certain message is received on a certain topic.

Connect and subscribe

Then, we have the `connect_and_subscribe()` function. This function is responsible for connecting to the broker as well as to subscribe to a topic.

```
def connect_and_subscribe():
```

We start by declaring the `client_id`, `mqtt_server` and `topic_sub` variables as global variables. This way, we can access these variables throughout the code.

```
global client_id, mqtt_server, topic_sub
```

Then, create an `MQTTClient` object called `client`. We need to pass as parameters the `client_id`, and the IP address of the MQTT broker (`mqtt_server`). These variables were set on the *boot.py* file.

```
client = MQTTClient(client_id, mqtt_server)
```

After that, set the callback function to the client (`sub_cb`).

```
client.set_callback(sub_cb)
```

Next, connect the client to the broker using the `connect()` method on the `MQTTClient` object.

```
client.connect()
```

After connecting, subscribe to the `topic_sub` topic. We set the `topic_sub` in the *boot.py* file (***notification***).

```
client.subscribe(topic_sub)
```

Finally, print a message with debugging information and return the `client`.

```
print('Connected to %s MQTT broker, subscribed to %s topic' %
(mqtt_server, topic_sub))
return client
```

Restart and reconnect

We create a function called `restart_and_reconnect()`. This function will be called in case the ESP32 or ESP8266 fails to connect to the broker.

This function prints a message to inform that the connection was not successful.

We wait 10 seconds. Then, reset the ESP using the `reset()` method.

```
def restart_and_reconnect():
  print('Failed to connect to MQTT broker. Reconnecting...')
  time.sleep(10)
  machine.reset()
```

Receive and publish messages

Until now, we've created functions to handle tasks related with the MQTT communication. From now on, the code will call those functions to make things happen.

The first thing we need to do is to connect to the MQTT broker and subscribe to a topic. So, create a client by calling the `connect_and_subscribe()` function.

```
try:
  client = connect_and_subscribe()
```

In case we're not able to connect to the MQTTT broker, we'll restart the ESP by calling the `restart_and_reconnect()` function:

```
except OSError as e:
  restart_and_reconnect()
```

In the `while` loop is where we'll be receiving and publishing the messages. We use `try` and `except` statements to prevent the ESP from crashing in case something goes wrong.

Inside the try block, we start by applying the `check_msg()` method in the `client`.

```
try:
  client.check_msg()
```

The `check_msg()` method checks whether a pending message from the server is available. It waits for a single incoming MQTT message and process it. The subscribed messages are delivered to the callback function we've defined earlier (the `sub_cb()` function). If there isn't a pending message, it returns `None`.

Then, add an `if` statement to check whether 5 seconds (`message_interval`) have passed since the last message was sent.

```
if (time.time() - last_message) > message_interval:
```

If it is time to send a new message, we create a `msg` variable with the "`Hello`" text followed by a counter.

```
msg = b'Hello #%d' % counter
```

To publish a message on a certain topic, you just need to apply the `publish()` method on the `client` and pass as arguments, the topic and the message. The `topic_pub` variable was set to ***hello*** in the *boot.py* file.

```
client.publish(topic_pub, msg)
```

After sending the message, update the last time a message was received by setting the `last_message` variable to the current time.

```
last_message = time.time()
```

Finally, we increase the counter variable in every loop.

```
counter += 1
```

If something unexpected happens, call the `restart_and_reconnect()` function.

```
except OSError as e:
  restart_and_reconnect()
```

That's it for ESP#1. Remember that you need to upload all these three to make the project work (you should upload the files in this order):

1. *umqttsimple.py*
2. *boot.py*
3. *main.py*

After uploading all files, you should get success messages on: establishing a network connection; connecting to the broker; and subscribing to the topic.

ESP #2

Let's now prepare ESP#2:

- It is subscribed to the ***hello*** topic
- It publishes on the ***notification*** topic

Like the ESP#1, you also need to upload the *umqttsimple.py*, *boot.py*, and *main.py* files.

Importing umqtttsimple

To use MQTT with the ESP32/ESP8266 and MicroPython, you need to install the `umqttsimple` library.

You can access the `umqttsimple` library code in the following link:

https://github.com/RuiSantosdotme/ESP-MicroPython/blob/master/code/MQTT/umqttsimple.py

boot.py

Copy the following code to the ESP#2 *boot.py* file.

```
import time
from umqttsimple import MQTTClient
import ubinascii
import machine
import micropython
import network
import esp
esp.osdebug(None)
import gc
gc.collect()

ssid = 'REPLACE_WITH_YOUR_SSID'
password = 'REPLACE_WITH_YOUR_PASSWORD'
mqtt_server = 'REPLACE_WITH_YOUR_MQTT_BROKER_IP'
#EXAMPLE IP ADDRESS
#mqtt_server = '192.168.1.144'
client_id = ubinascii.hexlify(machine.unique_id())
topic_sub = b'hello'
topic_pub = b'notification'
```

```
station = network.WLAN(network.STA_IF)

station.active(True)
station.connect(ssid, password)

while station.isconnected() == False:
  pass

print('Connection successful')
print(station.ifconfig())
```

SOURCE CODE
https://github.com/RuiSantosdotme/ESP-MicroPython/blob/master/code/MQTT/MQTT_Hello_World/ESP_2/boot.py

This code is very similar with the previous one. You need to replace the following variables with your network credentials and the broker IP address.

```
ssid = 'REPLACE_WITH_YOUR_SSID'
password = 'REPLACE_WITH_YOUR_PASSWORD'
mqtt_server = 'REPLACE_WITH_YOUR_MQTT_BROKER_IP'
```

The only difference here is that we subscribe to the ***hello*** topic and publish on the ***notification*** topic.

```
topic_sub = b'hello'
topic_pub = b'notification'
```

main.py

Copy the following code to the ESP#2 *main.py* file.

```
def sub_cb(topic, msg):
  print((topic, msg))
def connect_and_subscribe():
  global client_id, mqtt_server, topic_sub
  client = MQTTClient(client_id, mqtt_server)
  client.set_callback(sub_cb)
  client.connect()
  client.subscribe(topic_sub)
  print('Connected to %s MQTT broker, subscribed to %s topic' %
(mqtt_server, topic_sub))
  return client
def restart_and_reconnect():
  print('Failed to connect to MQTT broker. Reconnecting...')
```

```
  time.sleep(10)
  machine.reset()

try:
  client = connect_and_subscribe()
except OSError as e:
  restart_and_reconnect()

while True:
  try:
    new_message = client.check_msg()
    if new_message != 'None':
      client.publish(topic_pub, b'received')
    time.sleep(1)
  except OSError as e:
    restart_and_reconnect()
```

SOURCE CODE
https://github.com/RuiSantosdotme/ESP-MicroPython/blob/master/code/MQTT/MQTT_Hello_World/ESP_2/main.py

This code is very similar with the *main.py* from ESP#1.

We create the `sub_cb()`, the `connect_and_subscribe()` and the `restart_and_reconnect()` functions.

This time, the `sub_cb()` function just prints information about the topic and received message.

```
def sub_cb(topic, msg):
  print((topic, msg))
```

In the `while` loop, we check if we got a new message and save it in the `new_message` variable.

```
new_message = client.check_msg()
```

If we receive a new message, we publish a message saying 'received' on the `topic_sub` topic (in this case we set it to ***notification*** in the *boot.py* file).

```
if new_message != 'None':
  client.publish(topic_pub, b'received')
```

That's it for ESP#2. Remember that you need to upload all the next files to make the project work (you should upload the files in order):

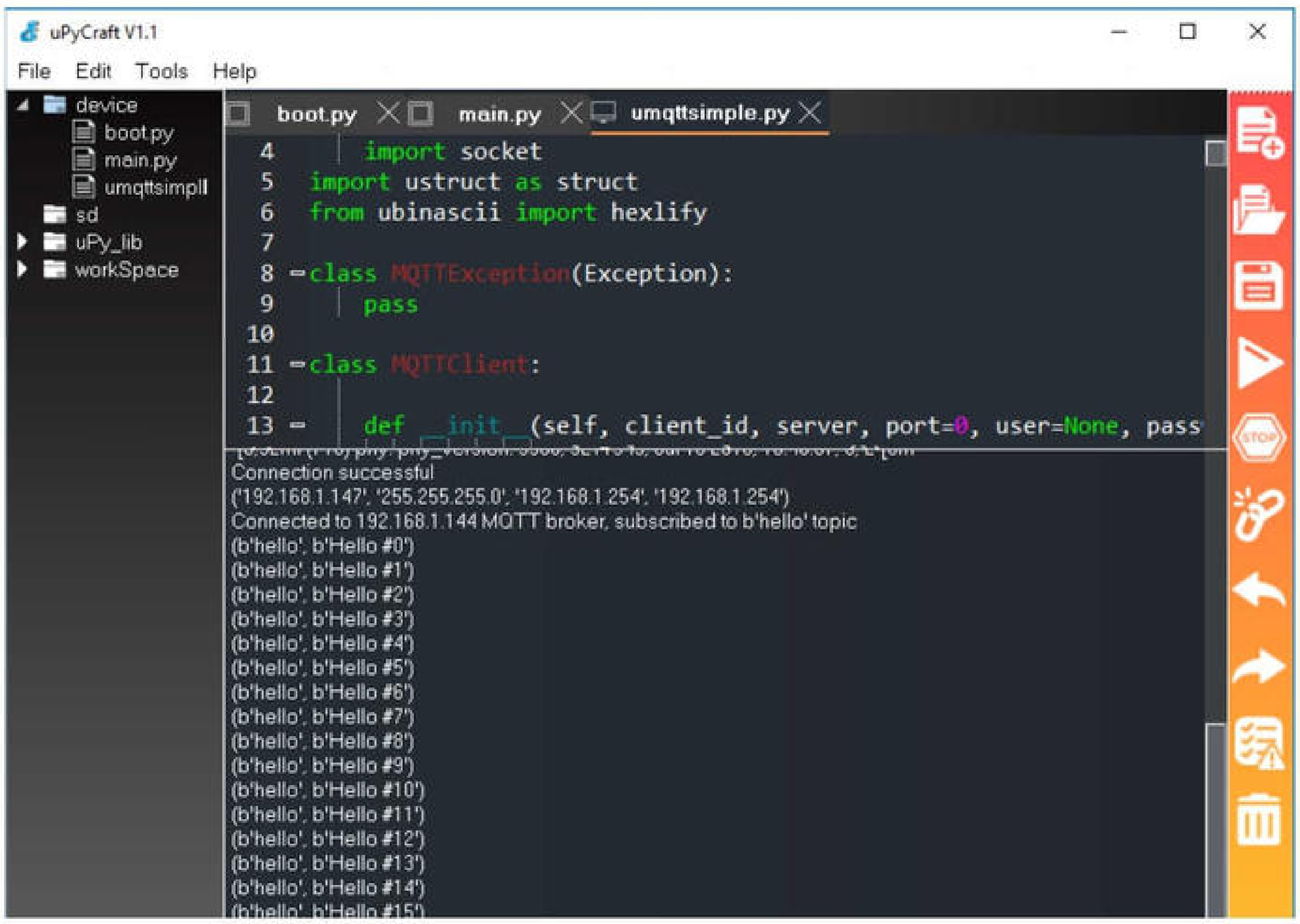
1. *umqttsimple.py*
2. *boot.py*
3. *main.py*

The ESP32/ESP8266 should establish a network connection and connect to the broker successfully.

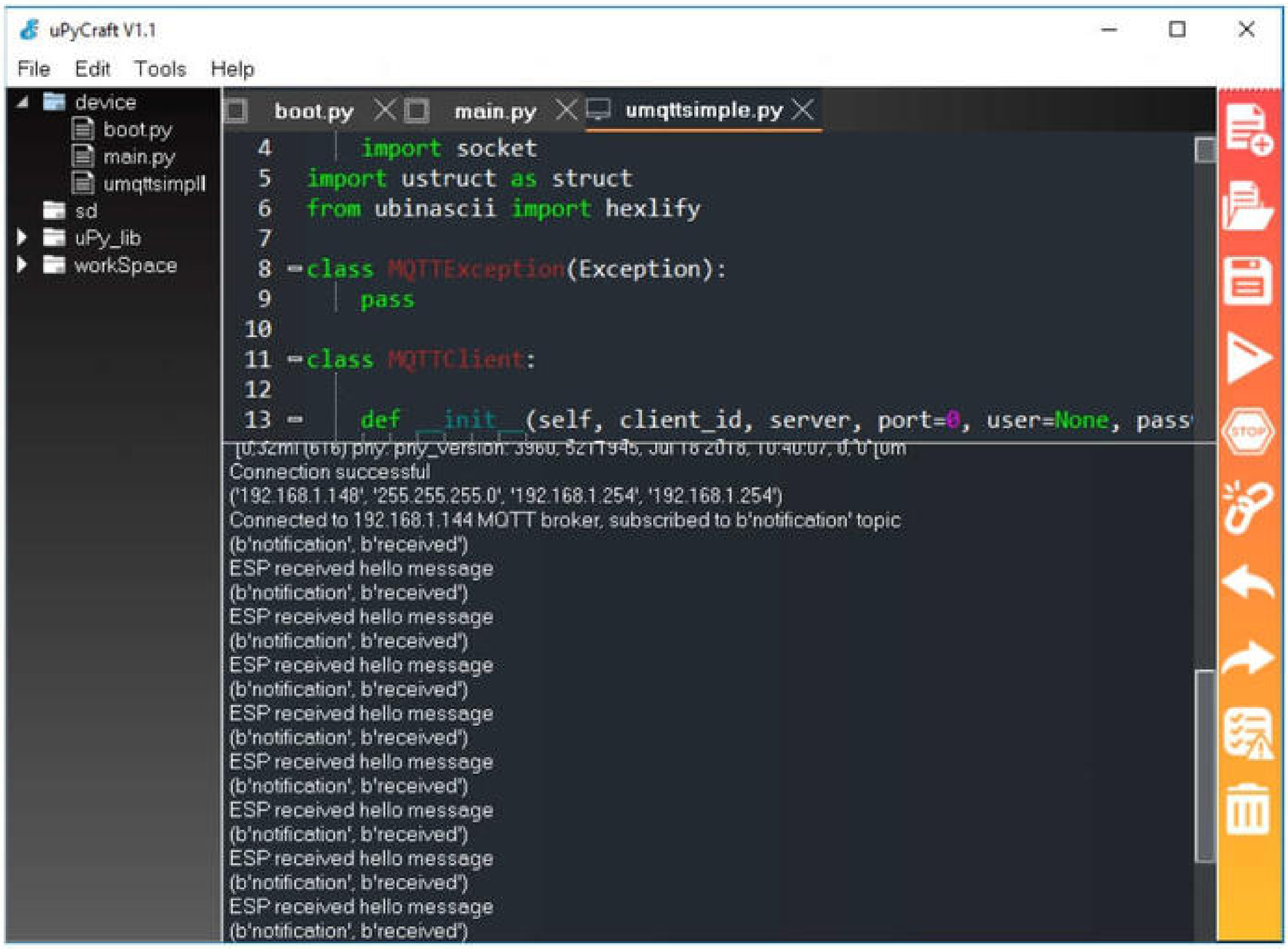
Demonstration

After uploading all the necessary scripts to both ESP boards and having both boards and the Raspberry Pi with the Mosquitto broker running, you are ready to test the setup.

The ESP#2 should be receiving the "Hello #" messages from ESP#1, as shown in the figure below.



On the other side, ESP#1 board should receive the "received" message. The "received" message is published by ESP#2 in the ***notification*** topic. ESP#1 is subscribed to that topic, so it receives the message.

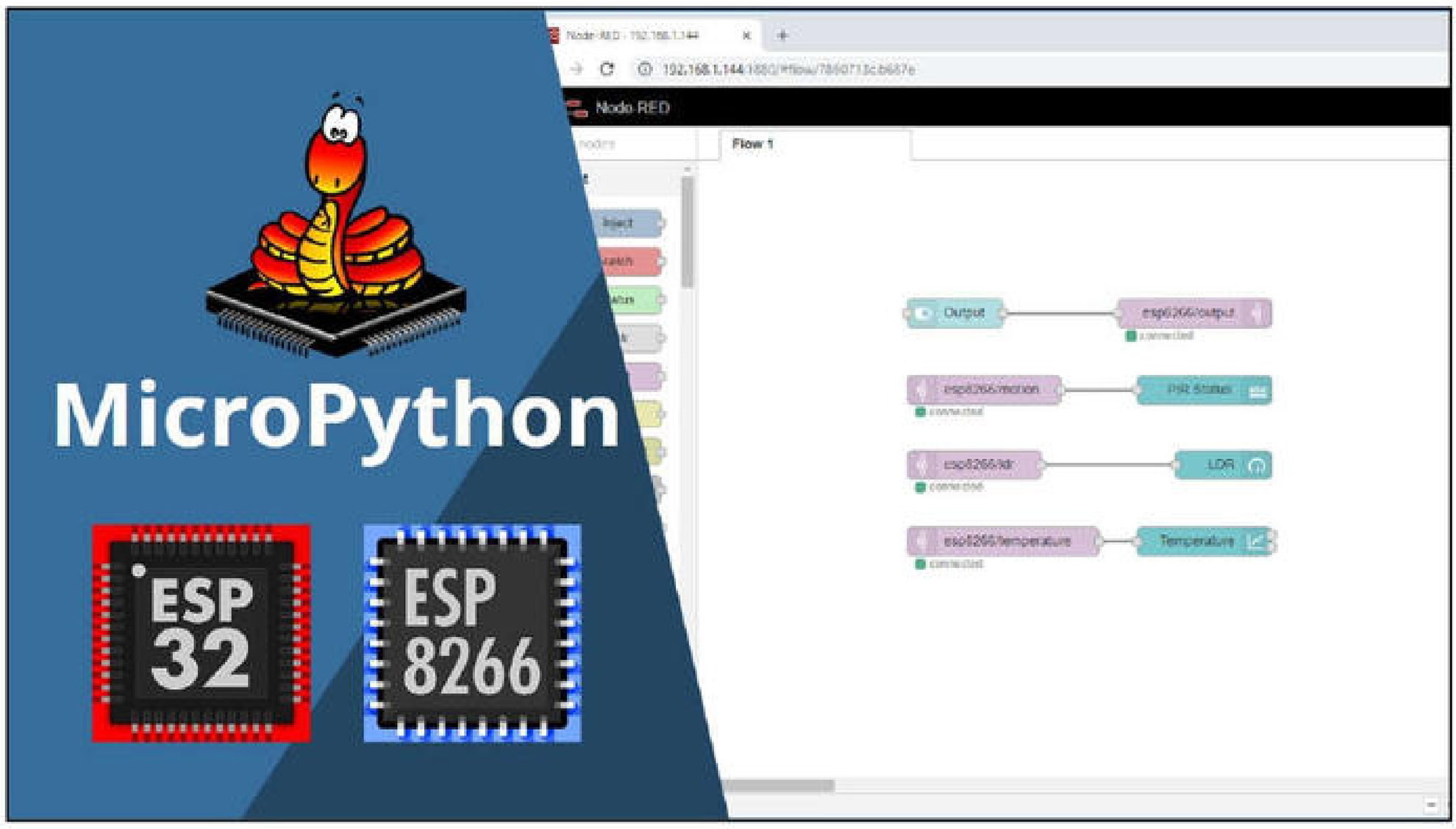


Wrapping Up

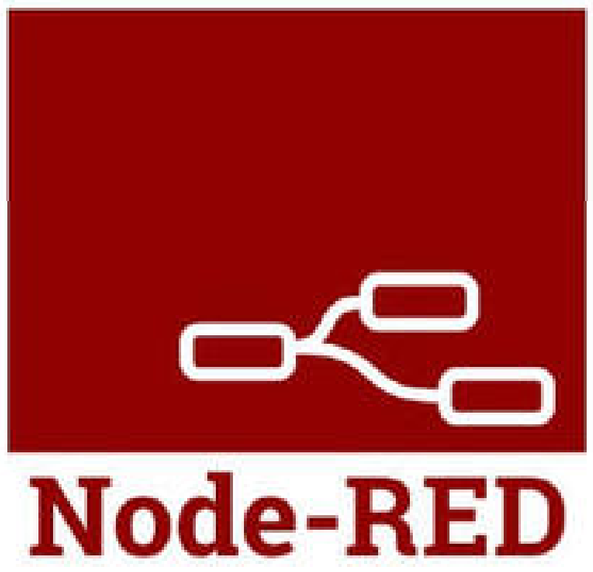
In this simple example, you've learned how to exchange text messages between two ESP32/ESP8266 boards using MQTT communication protocol. The idea is to use the concepts learned here to exchange useful data like sensor readings or commands to control outputs.

In the next Units, we'll show you how to exchange data between your ESP32/ESP8266 boards and Node-RED to display sensor readings and control outputs.

Installing Node-RED and Node-RED Dashboard on a Raspberry Pi



This Unit is a quick introduction to Node-RED. We'll cover what's Node-RED and how install Node-RED and Node-RED Dashboard nodes. If you want to communicate with Node-RED using your ESP32 or ESP8266 via MQTT, you need to be familiar with Node-RED first.



Note: building complex flows with Node-RED or explore all its functionalities is not the purpose of this Unit. With this Unit (and the next one) we intend to provide the necessary information on how to connect the ESP32/ESP8266 to Node-RED using MQTT communication protocol.

Prerequisites

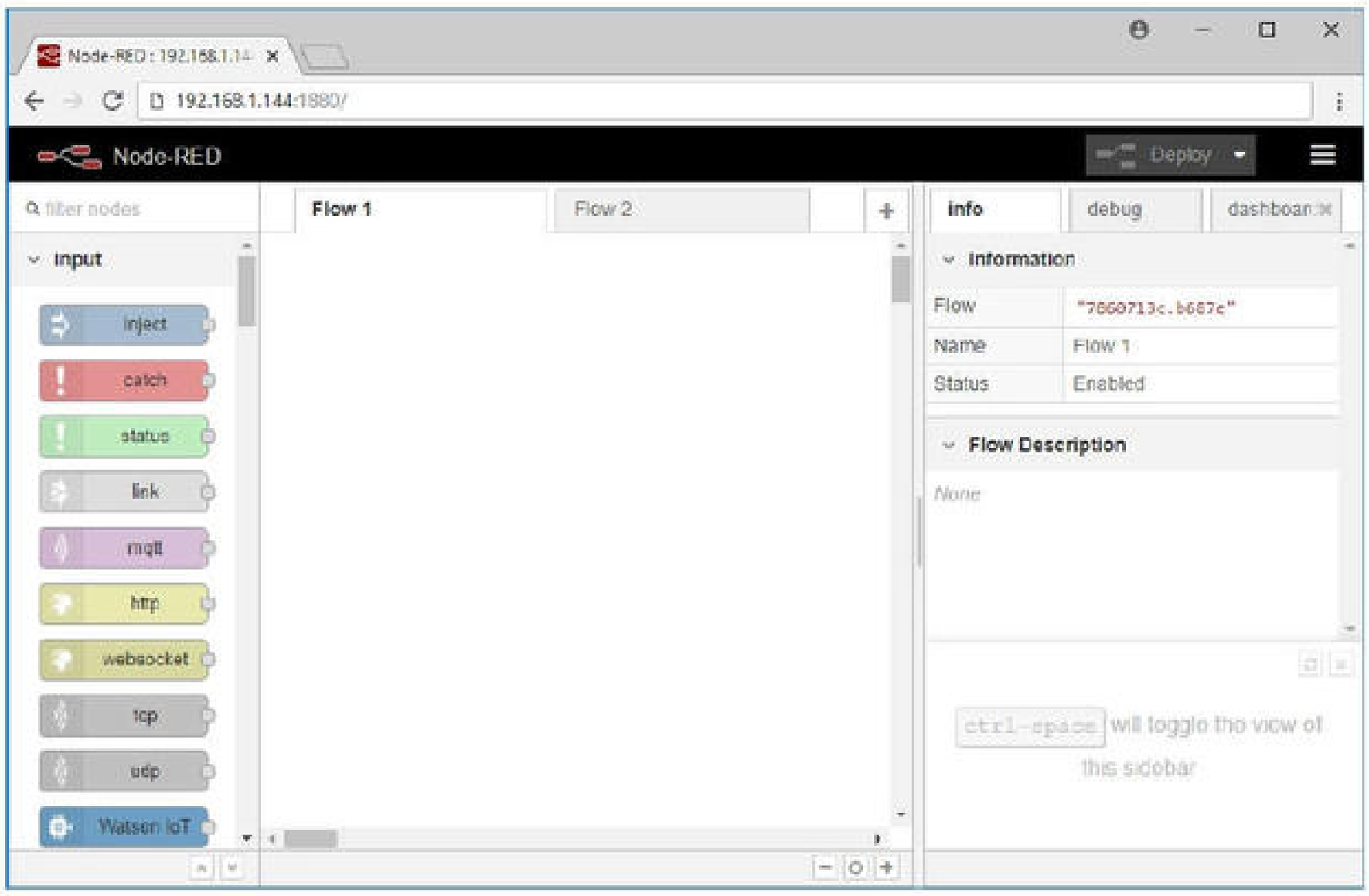
Before continuing:

- You should be familiar with the Raspberry Pi board – read Getting Started with Raspberry Pi;
- You should have the Raspbian or Raspbian Lite operating system installed in your Raspberry Pi – read Installing Raspbian Lite, Enabling and Connecting with SSH;
- You also need the following hardware: Raspberry Pi board – read Best Raspberry Pi Starter Kits, MicroSD Card – 16GB Class10, and Power Supply (5V 2.5A);
- You also need Mosquitto MQTT Broker installed (follow this Unit).

What's Node-RED?

Node-RED is a powerful open source visual wiring tool for building Internet of Things (IoT) applications.

Node-RED uses visual programming that allows you to connect code blocks, known as nodes, together to perform a task. The nodes when wired together are called flows.



Why Node-RED is a great solution?

- Node-RED is open source and developed by IBM and runs perfectly with the Raspberry Pi.
- Node-RED allows you to prototype a complex home automation system quickly, giving you more time to spend on designing and making cool stuff.

What can you do with Node-RED?

Node-RED makes it easy to:

- Access your Raspberry Pi GPIOs;
- Establish an MQTT connection with other boards (ESP32, ESP8266, Arduino, etc.);
- Create a responsive graphical user interface for your projects with Node-RED Dashboard;
- Communicate with third-party services (IFTTT.com, Adafruit.io, Thing Speak, and others);
- Retrieve data from the web (weather forecast, stock prices, emails, etc...);
- Create time triggered events;
- Store and retrieve data from a database;
- And much more...

Here's a resource with some flow examples and nodes for Node-RED.

Installing Node-RED

Installing Node-RED in your Raspberry Pi is quick and easy. It just takes a few commands.

Having a Terminal window of your Raspberry Pi open, enter the next command to install Node-RED:

```
pi@raspberry:~ $
bash <(curl -sL https://raw.githubusercontent.com/node-red/raspbian-deb-package/master/resources/update-nodejs-and-nodered)
```

The installation should be completed after a couple of minutes.

Autostart Node-RED on boot

To run Node-RED automatically when the Pi boots up, you need to enter the following command:

```
pi@raspberry:~ $ sudo systemctl enable nodered.service
```

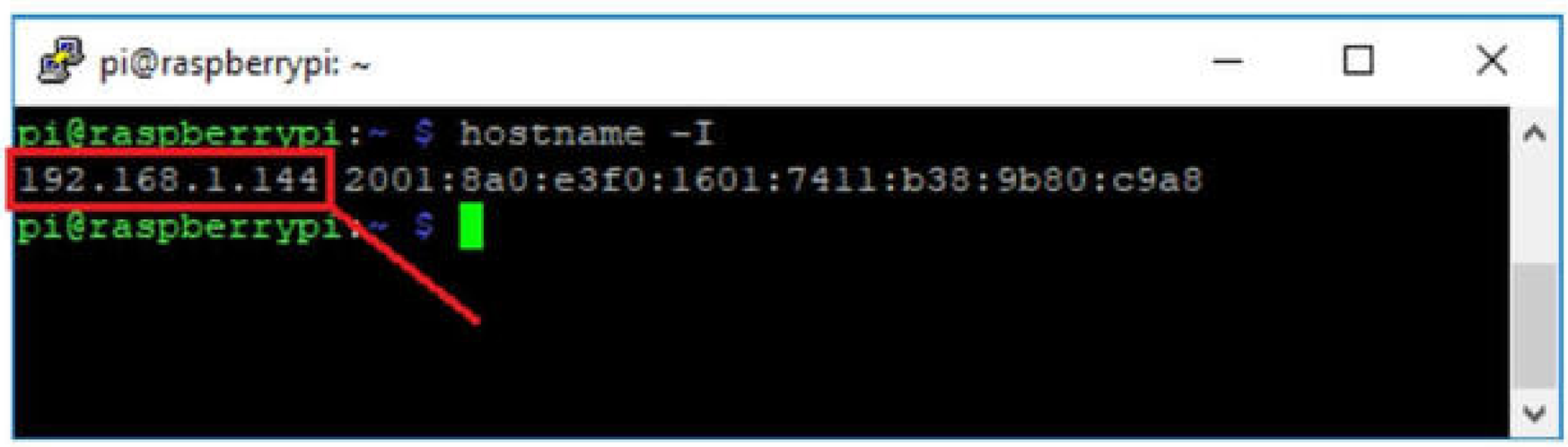
Now, restart your Raspberry Pi, so the auto-start takes effect:

```
pi@raspberry:~ $ sudo reboot
```

Raspberry Pi IP Address

To retrieve your Raspberry Pi IP address, type the next command in your Terminal window:

```
pi@raspberry:~ $ hostname -I
```



In our case, the Raspberry Pi IP address is 192.168.1.144 (save yours, because you'll need it in the next Unit).

Testing the Installation

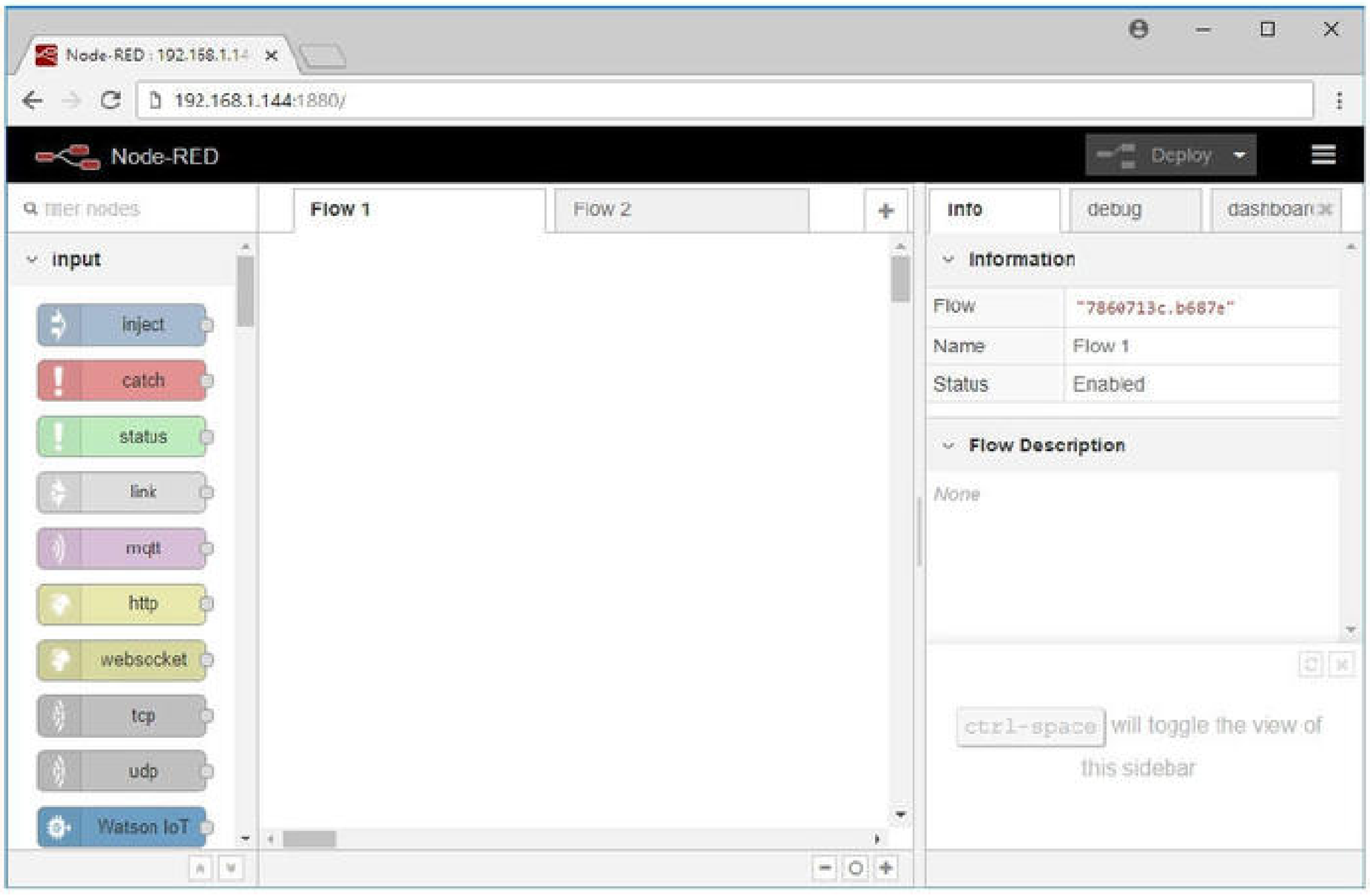
When your Pi is back on, you can test the installation by entering the IP address of your Pi in a web browser followed by the 1880 port number:

```
http://YOUR_RPi_IP_ADDRESS:1880
```

In my case is:

```
http://192.168.1.144:1880
```

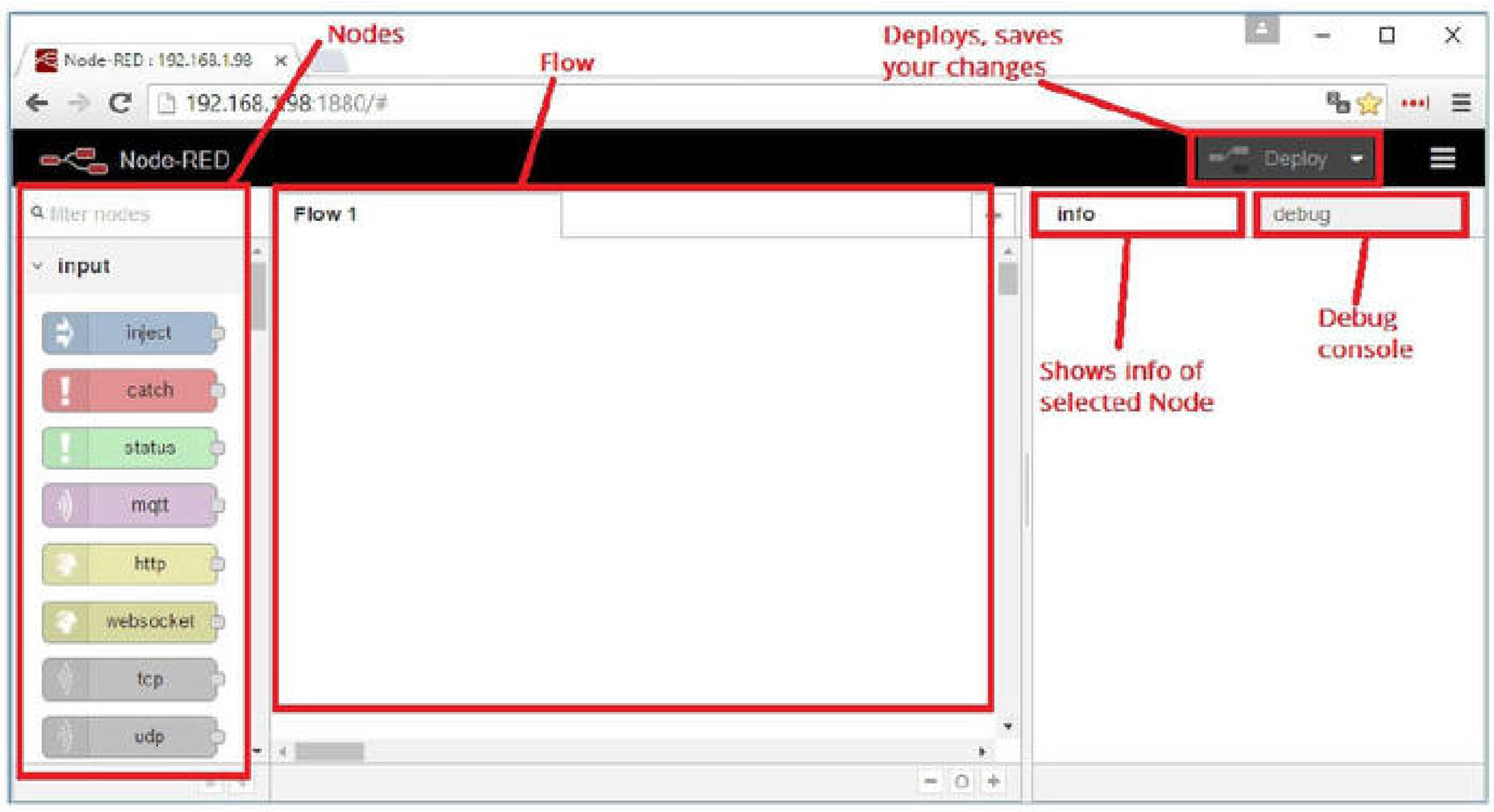
A page as shown in the next figure should load:



Node-RED Overview

On the left-side, you can see a list with a bunch of blocks. These blocks are called nodes and they are grouped by their functionality. If you select a node, you can see how it works in the info tab.

In the center, you have the Flow. The Flow is where you place the nodes.



Installing Node-RED Dashboard

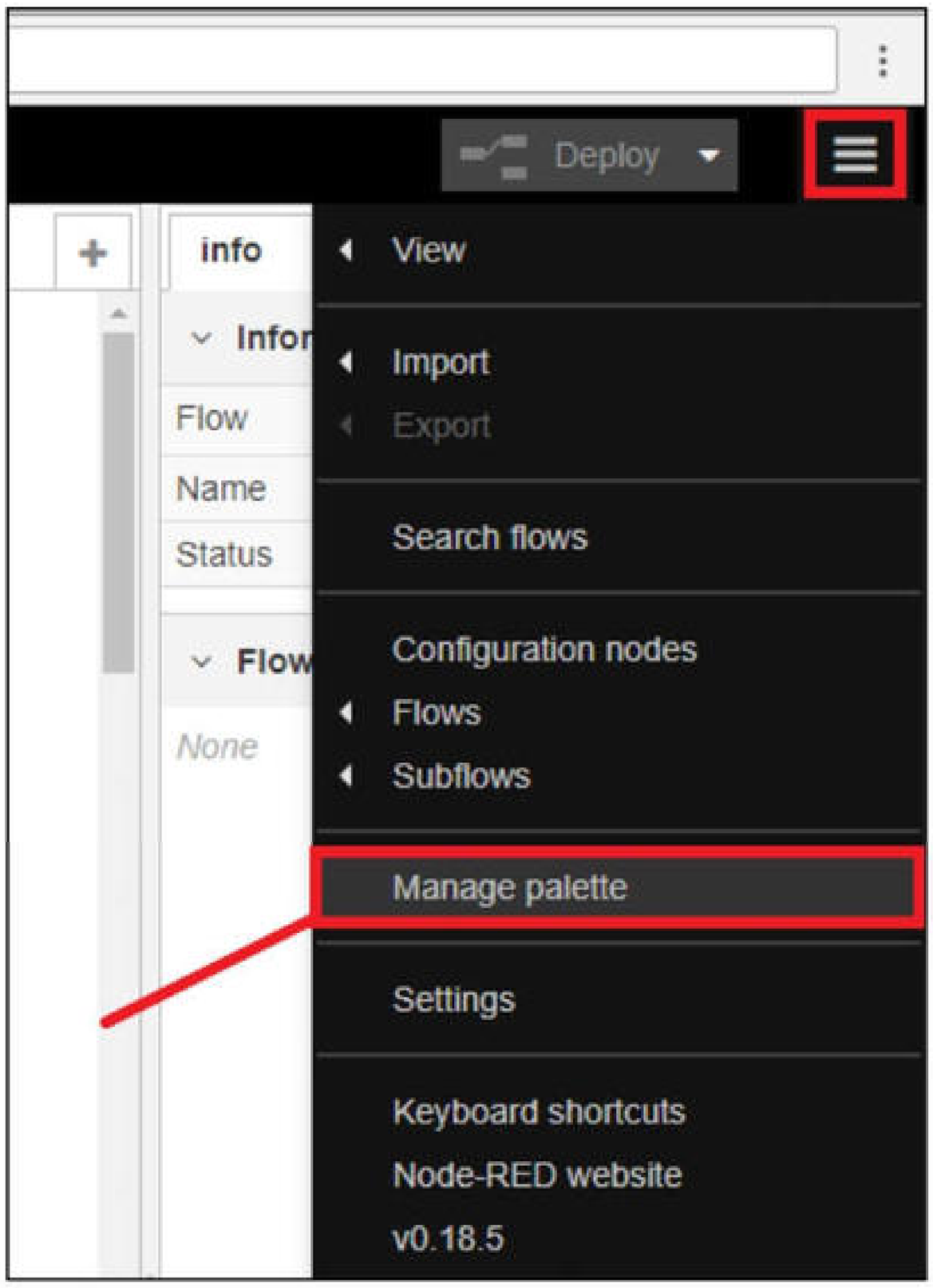
Node-RED Dashboard is a set of nodes that provides tools to create a user interface with buttons, charts, text, and other widgets.

To learn more about Node-RED Dashboard you can check the following links:

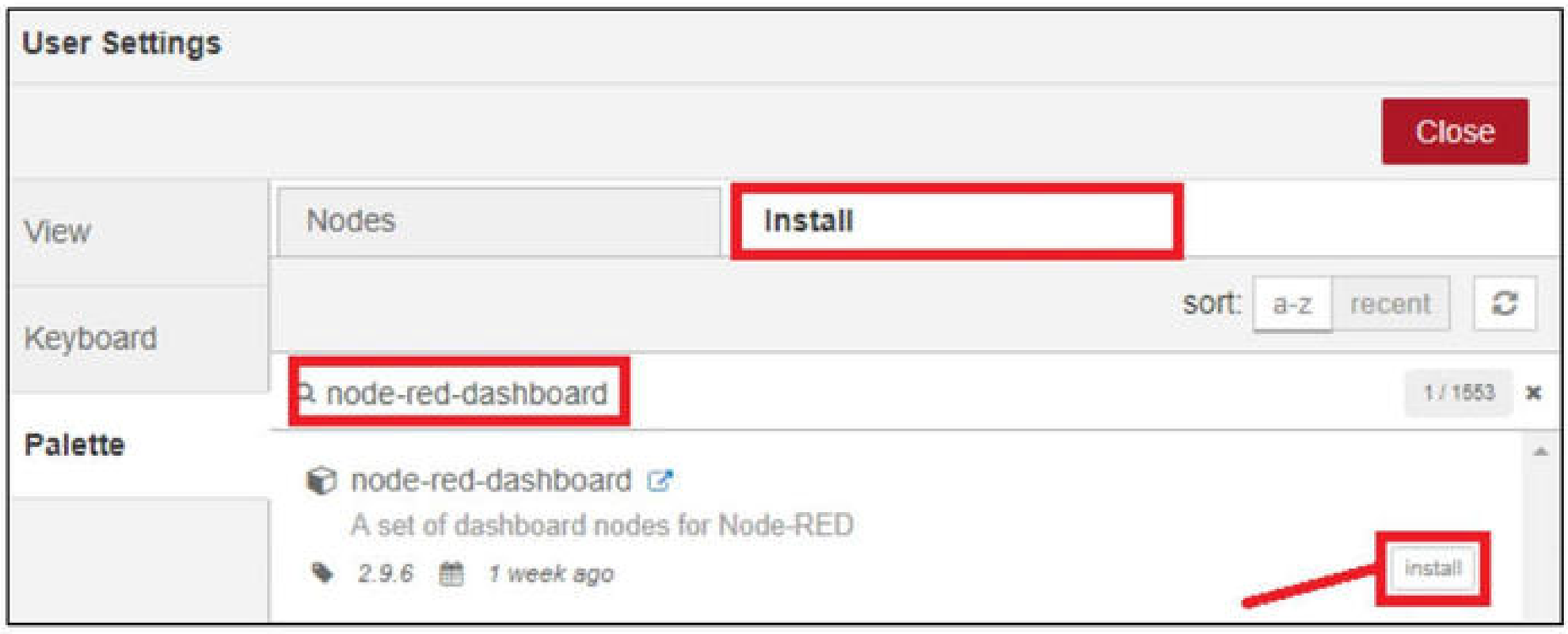
- Node-RED site: http://flows.nodered.org/node/node-red-dashboard
- GitHub: https://github.com/node-red/node-red-dashboard

Installing Node-RED Dashboard

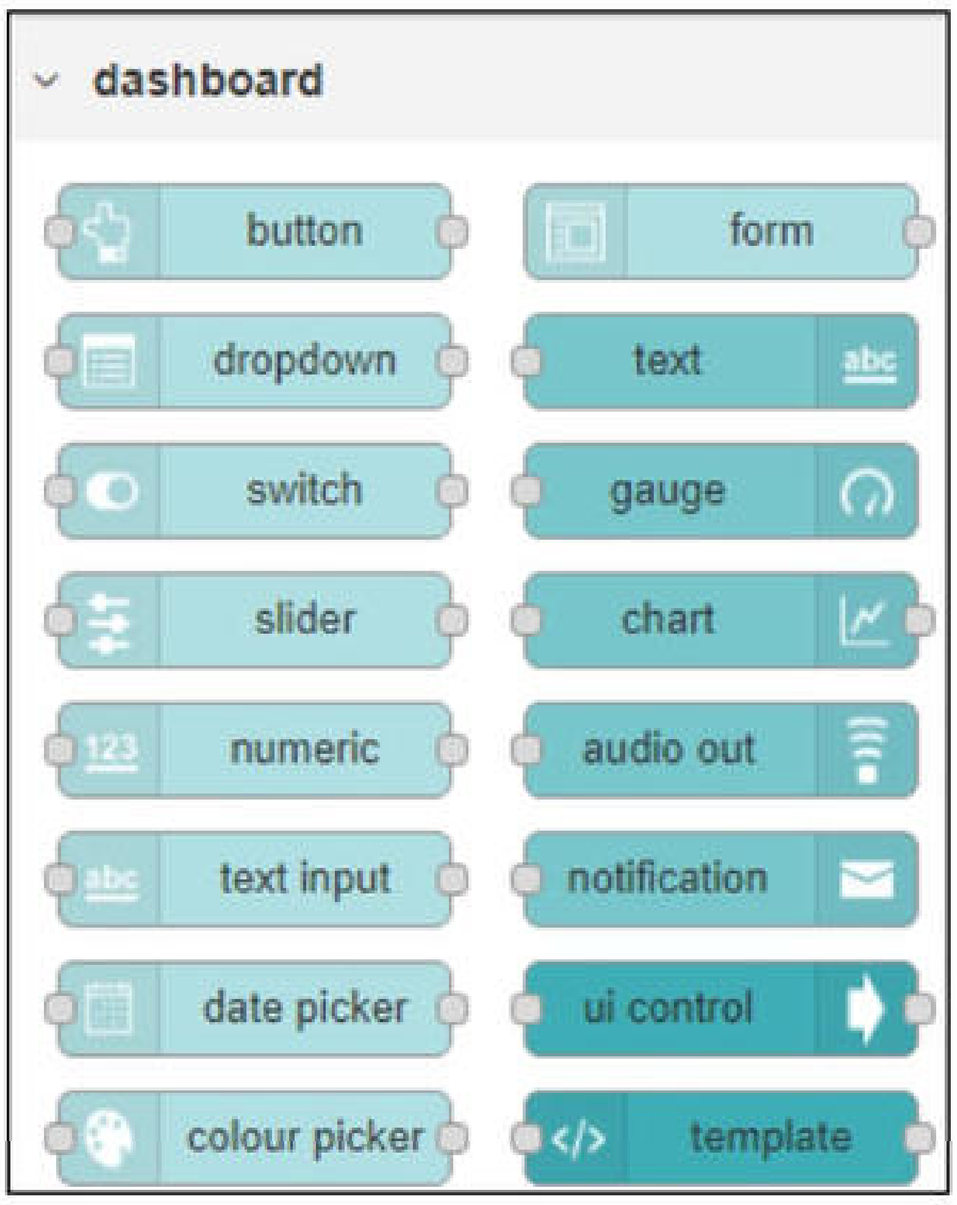
To install the Node-RED Dashboard, go to the "**Settings**" menu in the top-right corner and open the "**Manage palette**" menu:



A new window opens with the User Settings. Open the "**Install**" tab, search for "**node-red-dashboard**", and press the "**install**" button:



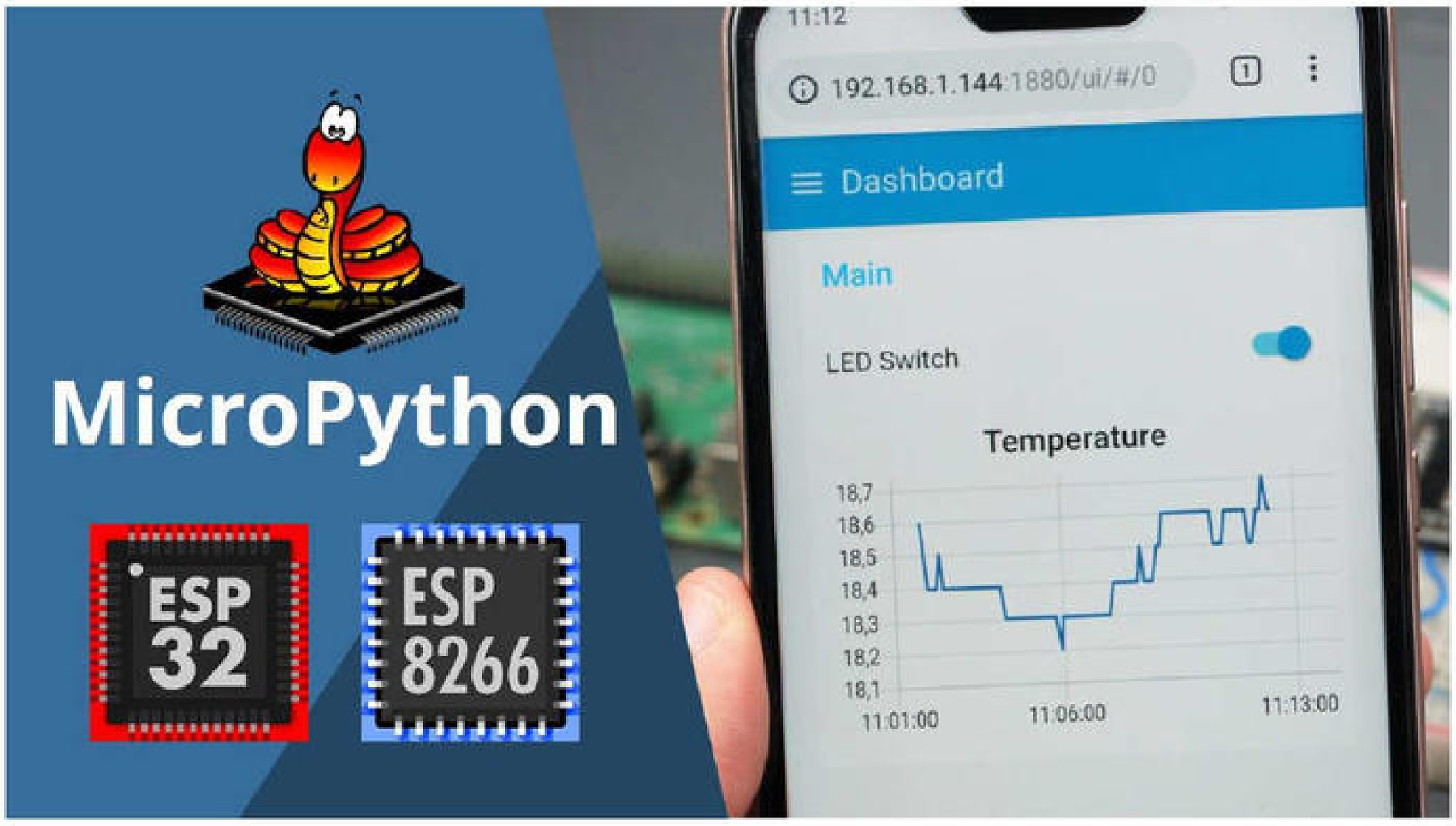
Close the menu and a new set of nodes should appear in your left window with the dashboard nodes:



Continue to the Next Unit

Now you have everything ready to integrate Node-RED with your ESP32 or ESP8266 and create a user interface. Continue to the next Unit to learn how to create a Node-RED user interface to interact with the ESP32/ESP8266 board using MQTT communication protocol.

MQTT - Connect ESP32/ESP8266 to Node-RED



In this Unit, we're going to demonstrate how to use Node-RED Dashboard to control the ESP32/ESP8266 GPIOs and display temperature readings in a chart. We're going to build a simple project to illustrate the most important concepts (publish and subscribe with Node-RED). You'll also learn how to measure temperature using the DS18B20 temperature sensor.

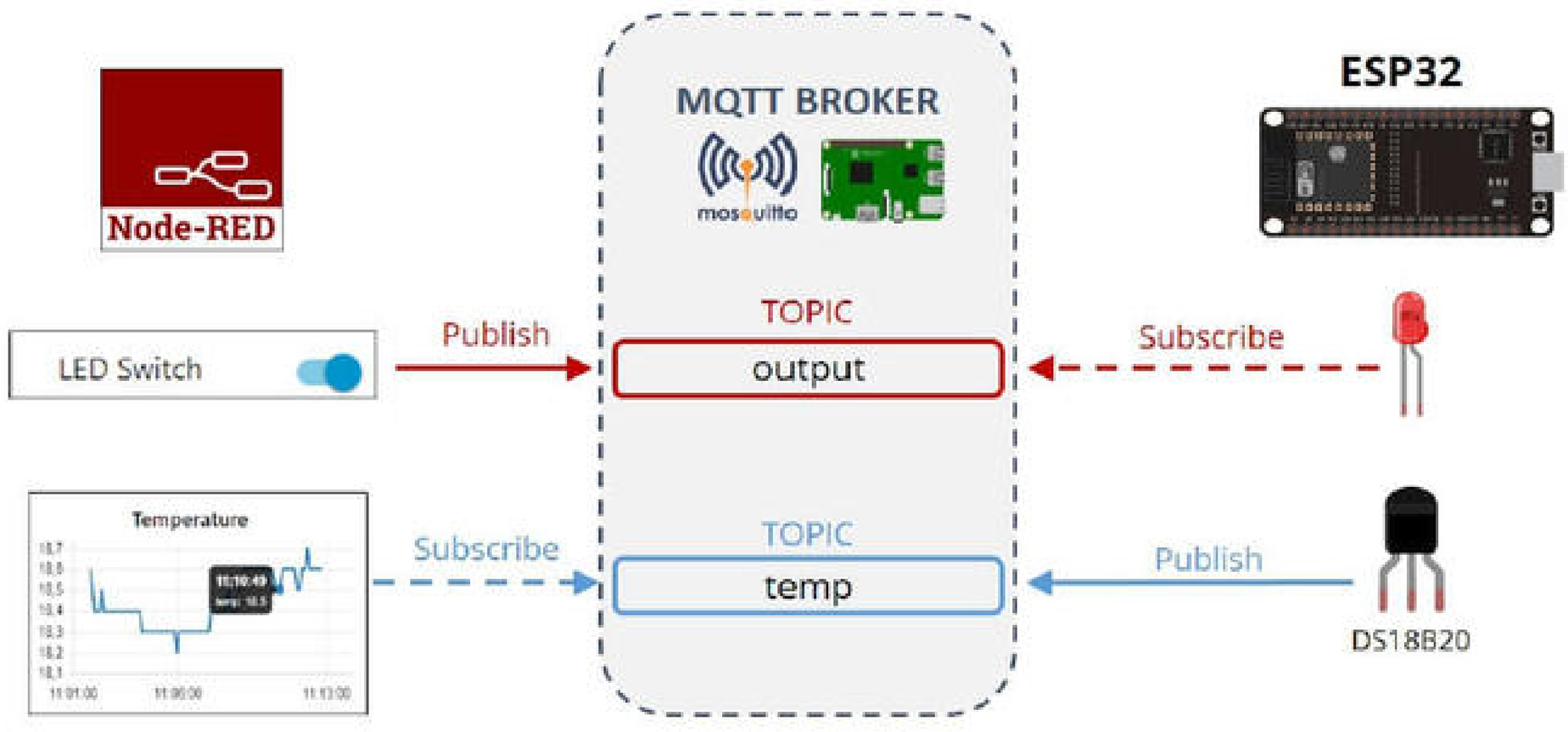
Project Overview

Here's a high-level overview of the project we'll build:

- In the Node-RED Dashboard, a slider switch allows you to control an output of the ESP32/ESP8266. To simplify the project, we'll control the on-board LED (GPIO 2).
- When you turn the slider switch on, Node-RED publishes a "on" message on the ***output*** topic. When the slider switch changes its state to off, it publishes a "off" message on the ***output*** topic.

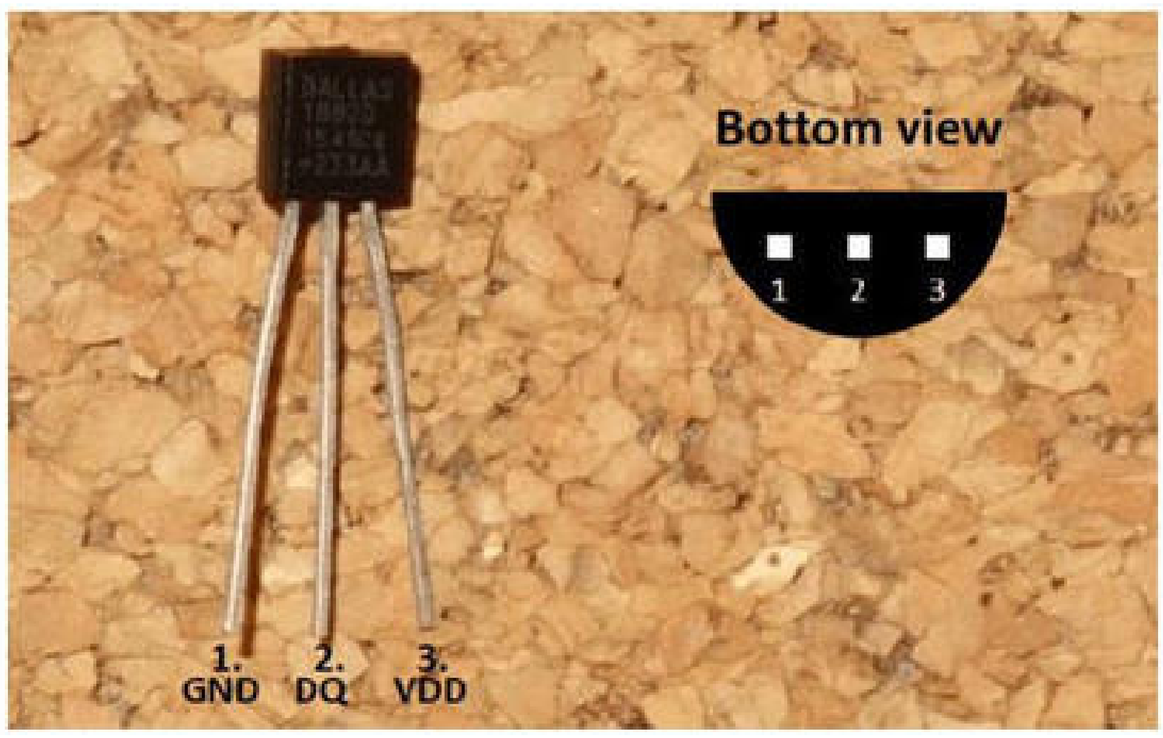
- The ESP32/ESP8266 is subscribed to the ***output*** topic. When it receives a message, it turns the LED on or off accordingly - instead of an LED you can control any other output on a different GPIO;
- The ESP32 takes temperature readings with the DS18B20 temperature sensor. The readings are published on the ***temp*** topic every 5 seconds;
- Node-RED is subscribed to the ***temp*** topic. So, it receives the DS18B20 temperature readings and publishes the readings in a chart.

The following figure shows the MQTT diagram of this setup:



Introducing the DS18B20 Temperature Sensor

The DS18B20 temperature sensor is a one-wire digital temperature sensor. This means that you can read the temperature with a very simple circuit setup. It communicates on common bus, which means that you can connect several devices and read their values using just one digital pin.



The DS18B20 is also available in waterproof version:

Features

Here's some main features of the DS18B20 temperature sensor:

- Communicates over 1-Wire bus communication
- Operating range temperature: -55ºC to 125ºC
- Accuracy +/-0.5 ºC (between the range -10ºC to 85ºC)

Schematic

For this project you need to wire the DS18B20 temperature sensor and an LED to the ESP32 or ESP8266.

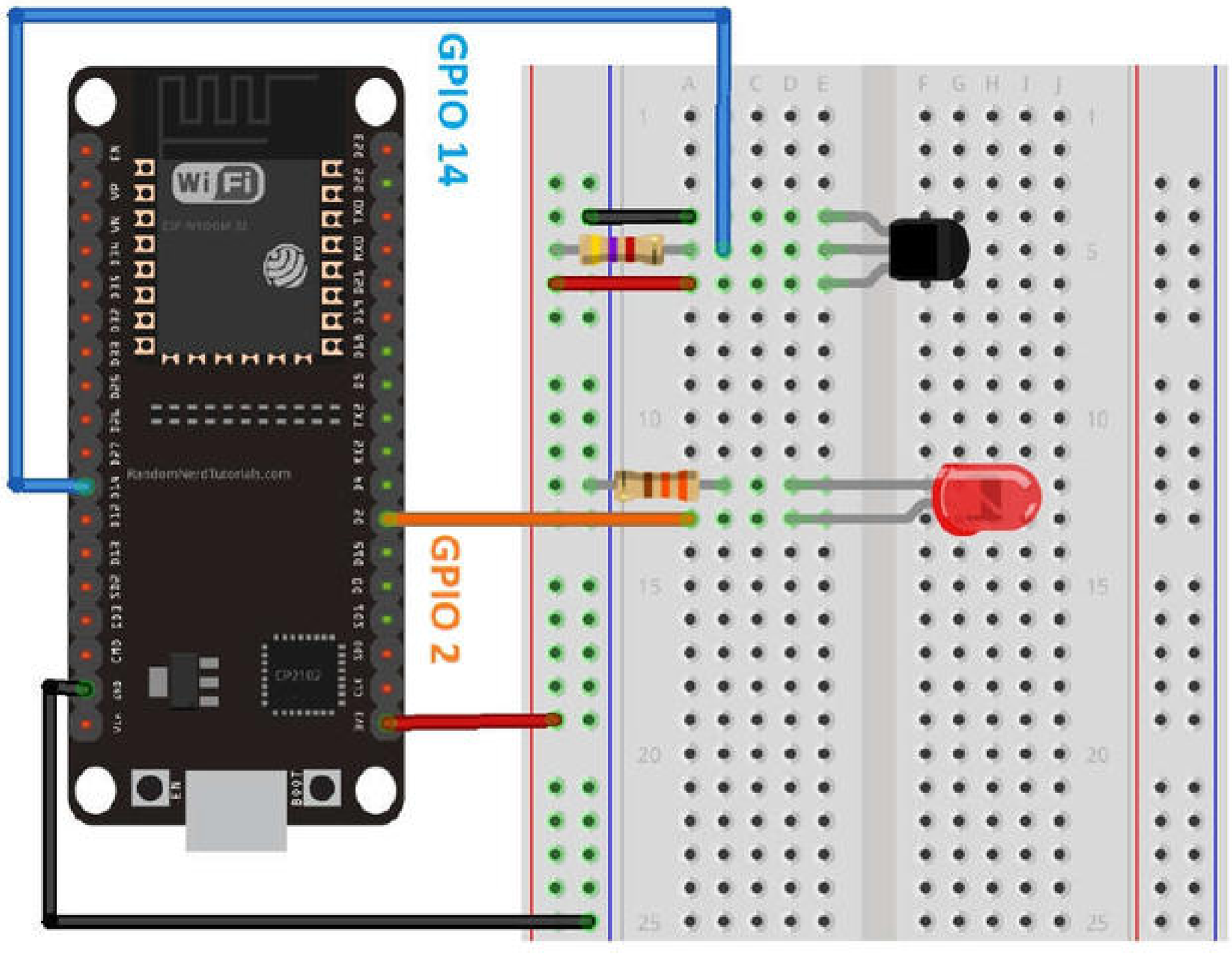
Parts required

Here's a list of parts you need to build the circuit:

- ESP32 or ESP8266
- 5mm LED
- 330 Ohm resistor
- DS18B20 temperature sensor
- Jumper wires

Schematic – ESP32

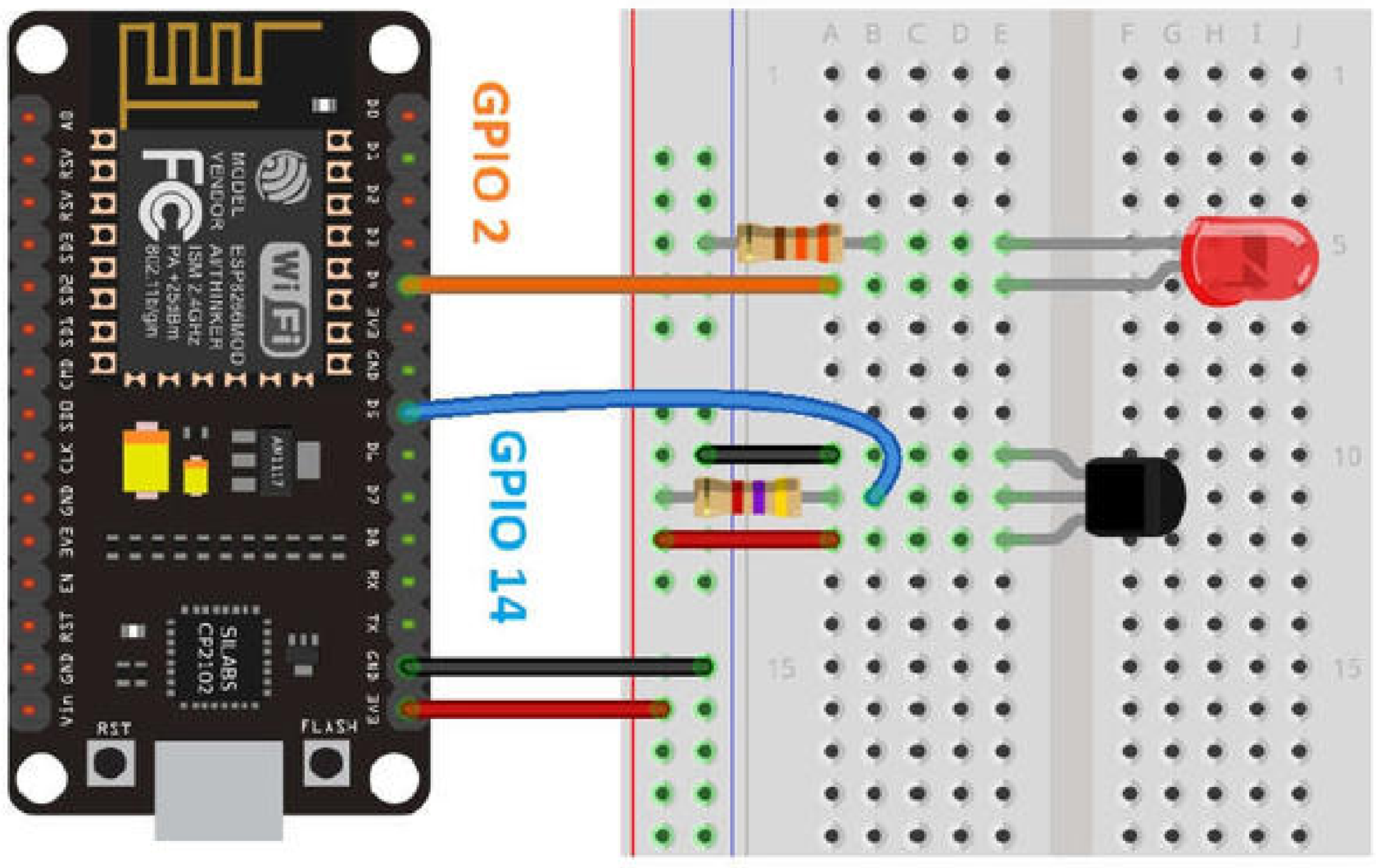
Follow the next schematic diagram if you're using an ESP32 board:



This schematic uses the ESP32 DEVKIT DOIT board version with 36 GPIOs. Before assembling the circuit double-check the pinout for the board you're using.

Schematic – ESP8266

Follow the next schematic diagram if you're using an ESP8266 board:



Importing *umqtttsimple*

To use MQTT with the ESP32/ESP8266 and MicroPython, you need to install the `umqttsimple` library.

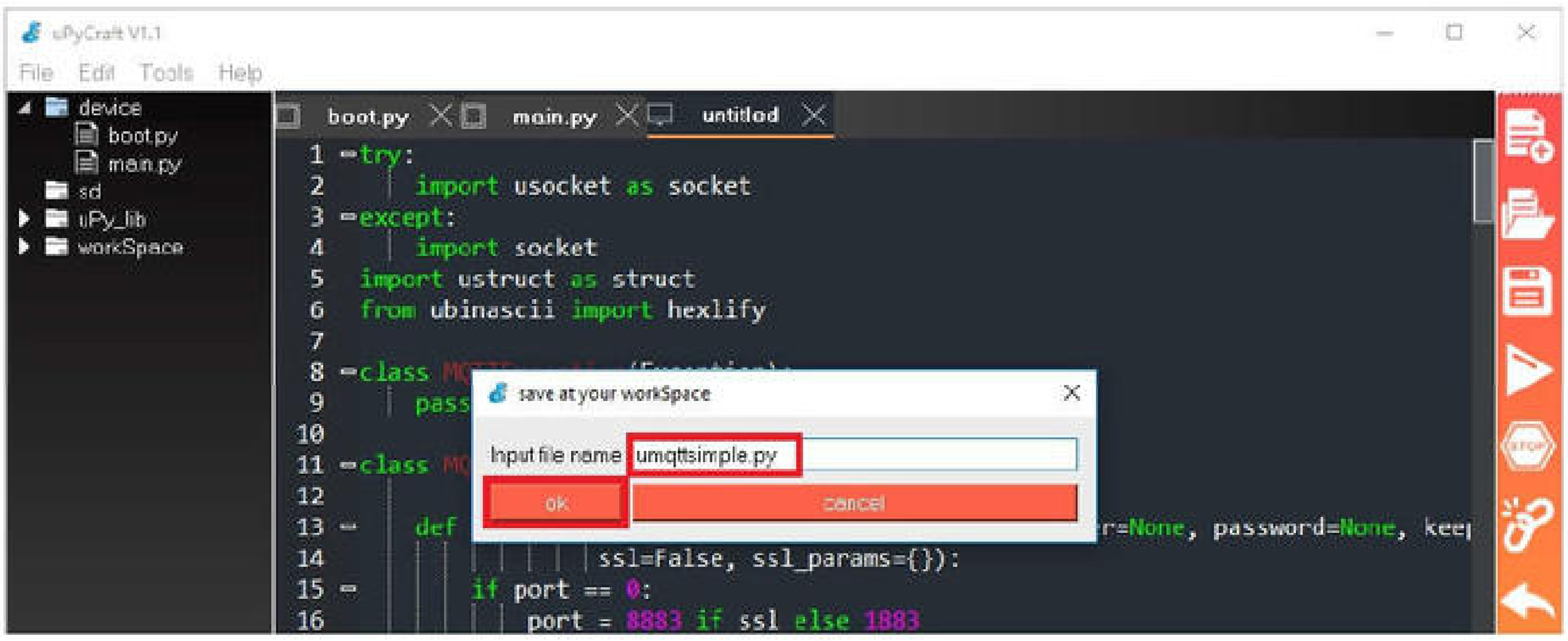
Create a new file by pressing the **New File** button.

Copy the `umqttsimple` library code into it. You can access the `umqttsimple` library code in the following link:

https://github.com/RuiSantosdotme/ESP-MicroPython/blob/master/code/MQTT/umqttsimple.py

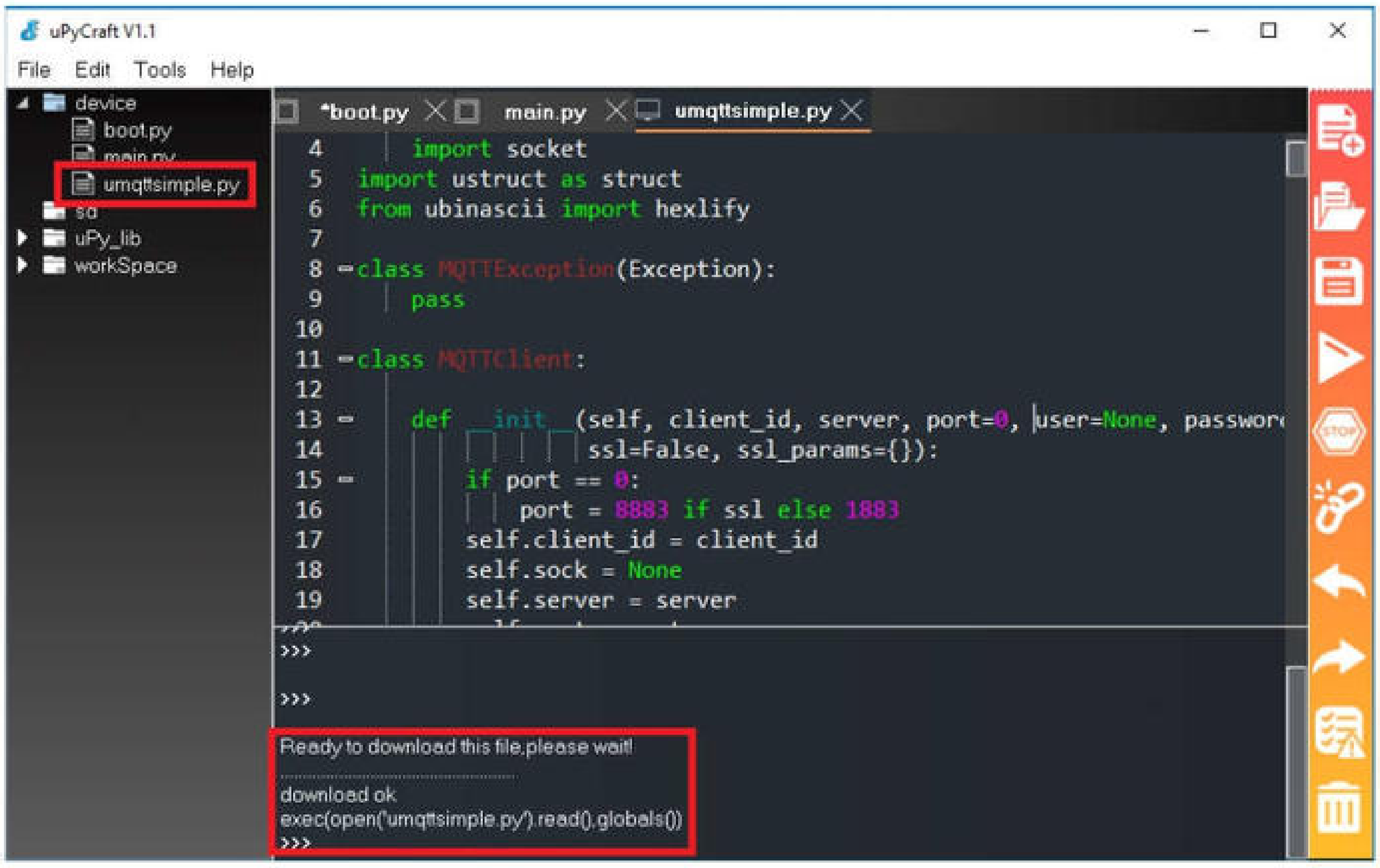
Save the file by pressing the **Save** button.

Call this new file "**umqttsimple.py**" and press **ok**.



Click the **Download and Run** button.

The file should be saved on the *device* folder with the name "**umqttsimple.py**" as highlighted in the figure below.



Now, you can use the library functionalities in your code by importing the library.

Note: if you're using Thonny IDE, you just need to:

1. Copy the library code to a new file.

2. Name that file *umqttsimple.py*.

3. Go to **Device ▸ Upload current script with the current name**.

boot.py

Copy the following code to the ESP32 or ESP8266 *boot.py* file.

```
import time
import onewire
import ds18x20
from umqttsimple import MQTTClient
import ubinascii
import machine
import micropython
import network
import esp
esp.osdebug(None)
import gc
gc.collect()
ssid = 'REPLACE_WITH_YOUR_SSID'
```

```
password = 'REPLACE_WITH_YOUR_PASSWORD'
mqtt_server = 'REPLACE_WITH_YOUR_MQTT_BROKER_IP'
#EXAMPLE IP ADDRESS
#mqtt_server = '192.168.1.144'
client_id = ubinascii.hexlify(machine.unique_id())
topic_sub = b'output'
topic_pub = b'temp'

last_sensor_reading = 0
readings_interval = 5

station = network.WLAN(network.STA_IF)

station.active(True)
station.connect(ssid, password)

while station.isconnected() == False:
  pass

print('Connection successful')
print(station.ifconfig())

ds_pin = machine.Pin(14)
ds_sensor = ds18x20.DS18X20(onewire.OneWire(ds_pin))

led = machine.Pin(2, machine.Pin.OUT, value=0)
```

SOURCE CODE
https://github.com/RuiSantosdotme/ESP-MicroPython/blob/master/code/MQTT/Node_RED_Client/boot.py

The *boot.py* file for this example is similar with the examples in previous MQTT Units. We'll just look at the relevant parts for this specific project.

Among other libraries, you need to import the `onewire` and the `ds18x20` library to read from the DS18B20 temperature sensor.

```
import onewire
import ds18x20
```

You also need to import the `MQTTClient` class from the `umqttsimple` library to set the ESP32/ESP8266 as an MQTT client.

```
from umqttsimple import MQTTClient
```

You need to include your network credentials as well as the broker IP address on the following variables:

```
ssid = 'REPLACE_WITH_YOUR_SSID'
password = 'REPLACE_WITH_YOUR_PASSWORD'
mqtt_server = 'REPLACE_WITH_YOUR_MQTT_BROKER_IP'
```

The ESP32/ESP8266 is subscribed to the ***output*** topic and publishes on the ***temp*** topic.

```
topic_sub = b'output'
topic_pub = b'temp'
```

We create auxiliary variables to save the last time a sensor reading was taken (`last_sensor_reading`) and to save the interval between readings (`readings_interval`). Here, we're setting the interval to 5 seconds.

```
last_sensor_reading = 0
readings_interval = 5
```

Create a variable called `ds_pin` that refers to GPIO 14, the pin the data wire of the DS18B20 temperature sensor is connected to.

```
ds_pin = machine.Pin(14)
```

Then, create a `ds18x20` object called `ds_sensor` on the `ds_pin` defined earlier. If you want to read the sensor using a different pin, you need to modify the previous line.

```
ds_sensor = ds18x20.DS18X20(onewire.OneWire(ds_pin))
```

Finally, create a `Pin` object on GPIO 2 called `led` that is off when the ESP32 first boots.

```
led = machine.Pin(2, machine.Pin.OUT, value=0)
```

That's it for the *boot.py* file. Proceed to the *main.py* file.

main.py

Copy the following code to the ESP32 or ESP8266 *main.py* file.

```
def read_ds_sensor():
  roms = ds_sensor.scan()
  print('Found DS devices: ', roms)
  print('Temperatures: ')
  ds_sensor.convert_temp()
  time.sleep(1)
  for rom in roms:
    temp = ds_sensor.read_temp(rom)
    if isinstance(temp, float):
      # uncomment for Fahrenheit
      #temp = temp * (9/5) + 32.0
      msg = (b'{0:3.1f}'.format(temp))
      print(temp, end=' ')
      print('Valid temperature')
      return msg
  return b'0.0'

def sub_cb(topic, msg):
  print((topic, msg))
  if msg == b'on':
    led.value(1)
  elif msg == b'off':
    led.value(0)

def connect_and_subscribe():
  global client_id, mqtt_server, topic_sub
  client = MQTTClient(client_id, mqtt_server)
  client.set_callback(sub_cb)
  client.connect()
  client.subscribe(topic_sub)
  print('Connected to %s MQTT broker, subscribed to %s topic' %
(mqtt_server, topic_sub))
  return client

def restart_and_reconnect():
  print('Failed to connect to MQTT broker. Reconnecting...')
  time.sleep(10)
  machine.reset()

try:
  client = connect_and_subscribe()
```

```
except OSError as e:
  restart_and_reconnect()

while True:
  try:
    client.check_msg()
    if (time.time() - last_sensor_reading) > readings_interval:
      msg = read_ds_sensor()
      client.publish(topic_pub, msg)
      last_sensor_reading = time.time()
  except onewire.OneWireError:
    print('Failed to read/publish sensor readings.')
    time.sleep(1)
  except OSError as e:
    restart_and_reconnect()
```

SOURCE CODE
https://github.com/RuiSantosdotme/ESP-MicroPython/blob/master/code/MQTT/Node_RED_Client/main.py

This script reads the temperature from the DS18B20 temperature sensor and publishes the sensor readings on the ***temp*** topic. It also subscribes the ESP32/ESP8266 to the ***output*** topic and turns the on-board LED on or off accordingly.

Let's take a closer look on how the code works.

DS18B20 temperature readings

We start by creating a function called `read_ds_sensor()`. You can use this function in other projects that use the DS18B20 sensor.

```
def read_ds_sensor():
```

The DS18B20 communicates via 1-Wire communication protocol. This means you can read several temperature sensors wired on the same GPIO.

The function starts by scanning for DS18B20 sensors and saves the found addresses on the `roms` variable.

```
roms = ds_sensor.scan()
```

Then, you need to execute the `convert_temp()` function to initiate a temperature reading and wait a second.

```
ds_sensor.convert_temp()
```

After that, we can read the temperature on the addresses found earlier by using the `read_temp()` method and passing the address as argument.

```
temp = ds_sensor.read_temp(rom)
```

Then, check if it has have valid temperature reading and create a `msg` variable with the sensor readings.

```
if isinstance(temp, float):
  # uncomment for Fahrenheit
  #temp = temp * (9/5) + 32.0
  msg = (b'{0:3.1f}'.format(temp))
```

By default, we're getting the temperature readings in Celsius degrees. If you want to get the temperature readings in Fahrenheit degrees, you need to uncomment the following line of code.

```
#temp = temp * (9/5) + 32.0
```

MQTT functions

Then, we create functions to handle MQTT tasks: `sub_cb`, `connect_and_subscribe`, and `restart_and_reconnect`. The `connect_and_subscribe`, and `restart_and_reconnect` functions were already explained in detail in previous Units.

The `sub_cb` function handles what happens when a new message is received on the topic we are subscribed to. In this case, the ESP32/ESP8266 is subscribed to the ***output*** topic. When it receives a new message, we check if the content is either "on" or "off" and set the LED state accordingly.

```
print((topic, msg))
if msg == b'on':
  led.value(1)
elif msg == b'off':
  led.value(0)
```

In the following `try` and `except` statement, we create a new client by calling the `connect_and_subscribe()` function. In case we're not able to create a new client, call the `restart_and_reconnect()` function.

```
try:
  client = connect_and_subscribe()
except OSError as e:
  restart_and_reconnect()
```

Getting and publishing messages

In the `while` loop, we check if we have a new message. New messages are then handled by the `sub_cb` function.

```
client.check_msg()
```

Then, check if 5 seconds (`readings_interval`) have passed since the last sensor reading.

```
if (time.time() - last_sensor_reading) > readings_interval:
```

If yes, call the `read_ds_sensor()` function to save a new temperature reading in the `msg` variable.

```
msg = read_ds_sensor()
```

After getting the temperature, publish it on the ***temp*** topic (`topic_sub`).

```
client.publish(topic_pub, msg)
```

Finally, update the last time the ESP32/ESP8266 published the temperature.

```
last_sensor_reading = time.time()
```

In case there's an error getting the temperature from the sensor, we print an error message, and wait for one second before trying again.

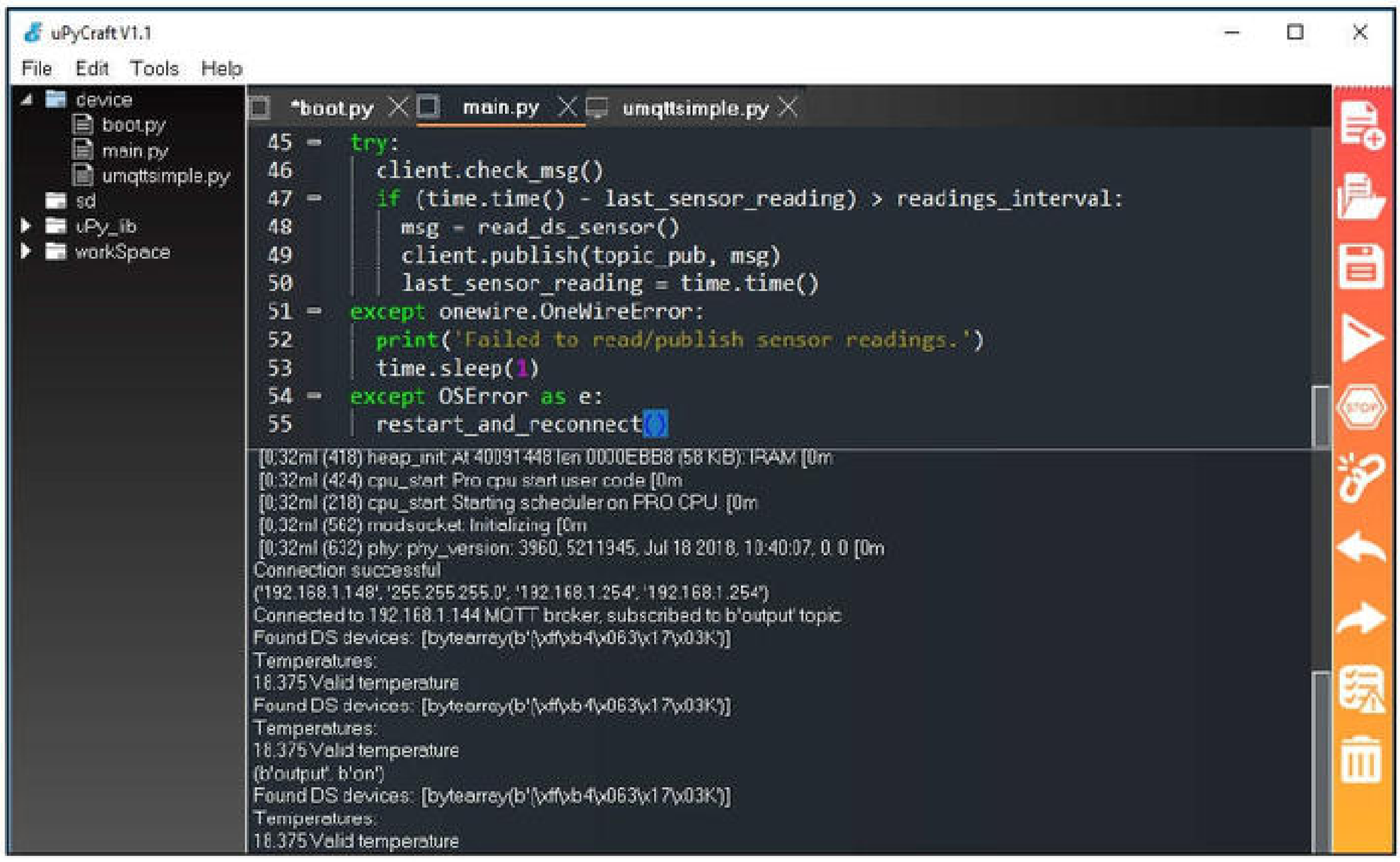
```
except onewire.OneWireError:
  print('Failed to read/publish sensor readings.')
  time.sleep(1)
```

Uploading the files

Upload all the provided files to the ESP32/ESP8266 in the following order:

1. *umqttsimple.py*
2. *boot.py*
3. *main.py*

The ESP32/ESP8266 should establish a successful connection to the broker and print sensor readings as shown in the following figure.



Creating the Node-RED flow

Before creating the flow, you need to have installed in your Raspberry Pi:

- Node-RED
- Node-RED Dashboard
- Mosquitto Broker

If you've followed previous Units, you should have everything installed and ready to proceed.

Starting Node-RED

To start Node-RED, enter the following in the Raspberry Pi Terminal window:

```
pi@raspberrypi:~ $ node-red-start
```

To access Node-RED, open a tab in any browser on the local network and type the following:

```
http://Your_RPi_IP_address:1880
```

You should replace **Your_RPi_IP_address** with your Raspberry Pi IP address. If you don't know your Raspberry Pi IP address, in the Terminal type:

```
pi@raspberrypi:~ $ hostname -I
```

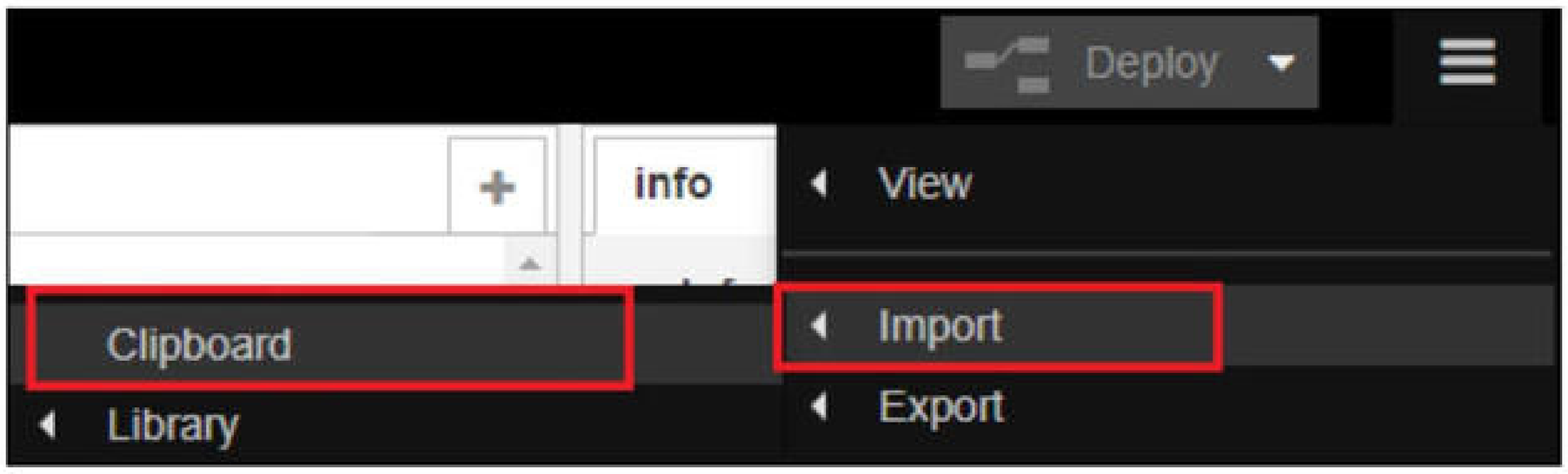
Importing the Node-RED flow

We won't show you how to create the Node-RED flow from scratch. We provide the code to import the complete flow, and then we'll take a closer look at each node and see what they do.

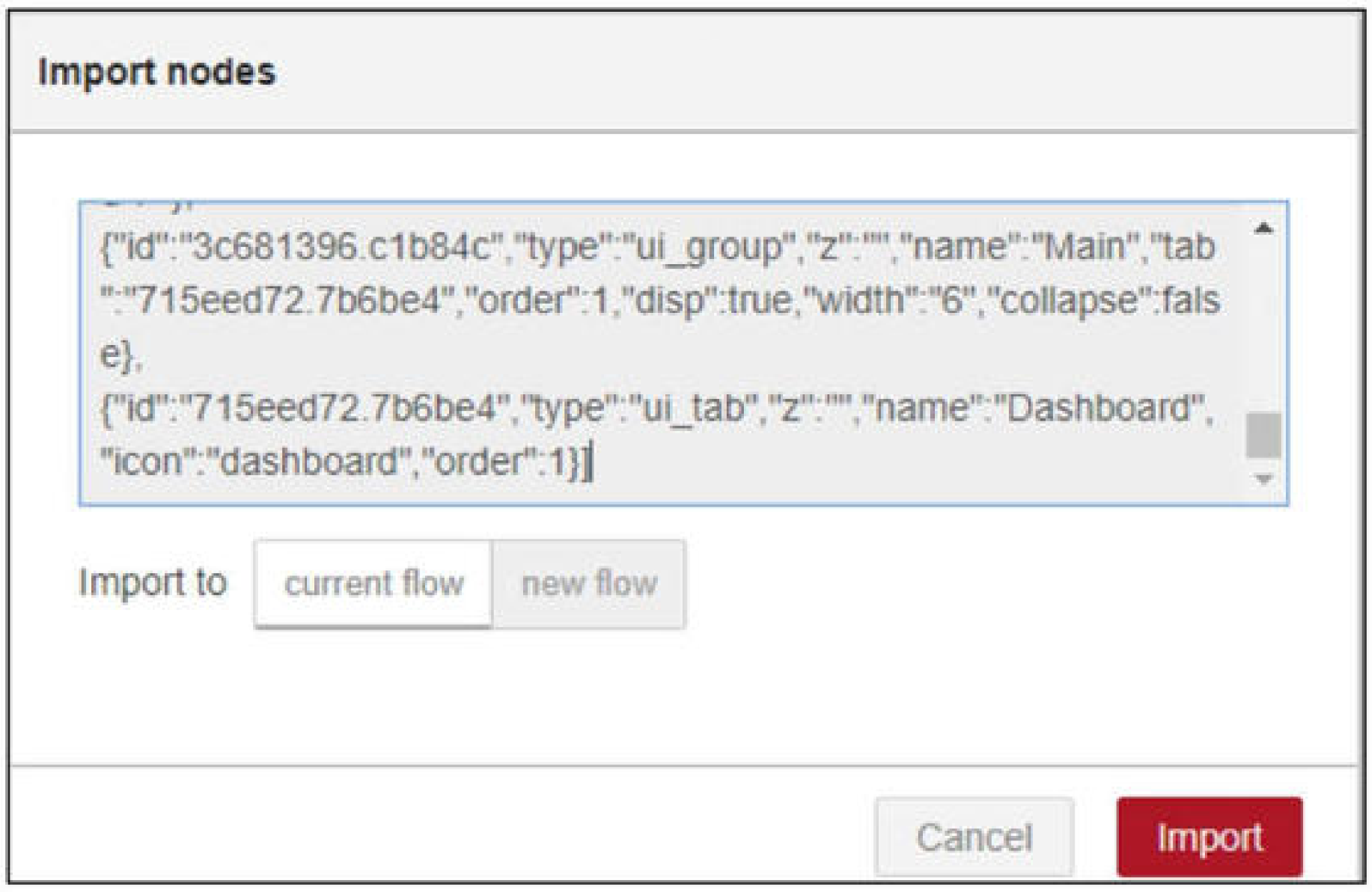
To import the Node-RED flow, go to the GitHub repository and copy the code provided:

Node-RED flow: https://raw.githubusercontent.com/RuiSantosdotme/ESP-MicroPython/master/code/MQTT/Node_RED_Client/Node_RED_Flow.txt

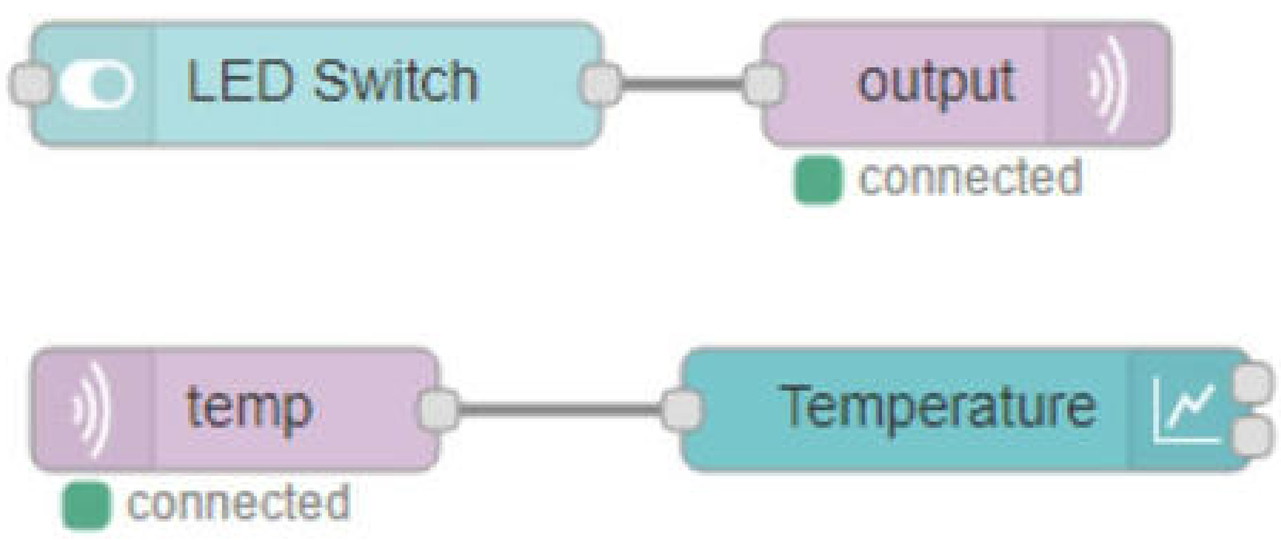
Next, in the Node-RED window, at the top right corner, select the menu, and go to **Import ▸ Clipboard**.



Then, paste the code provided and click **Import**.



After a successful import, you should see the following nodes on your flow.

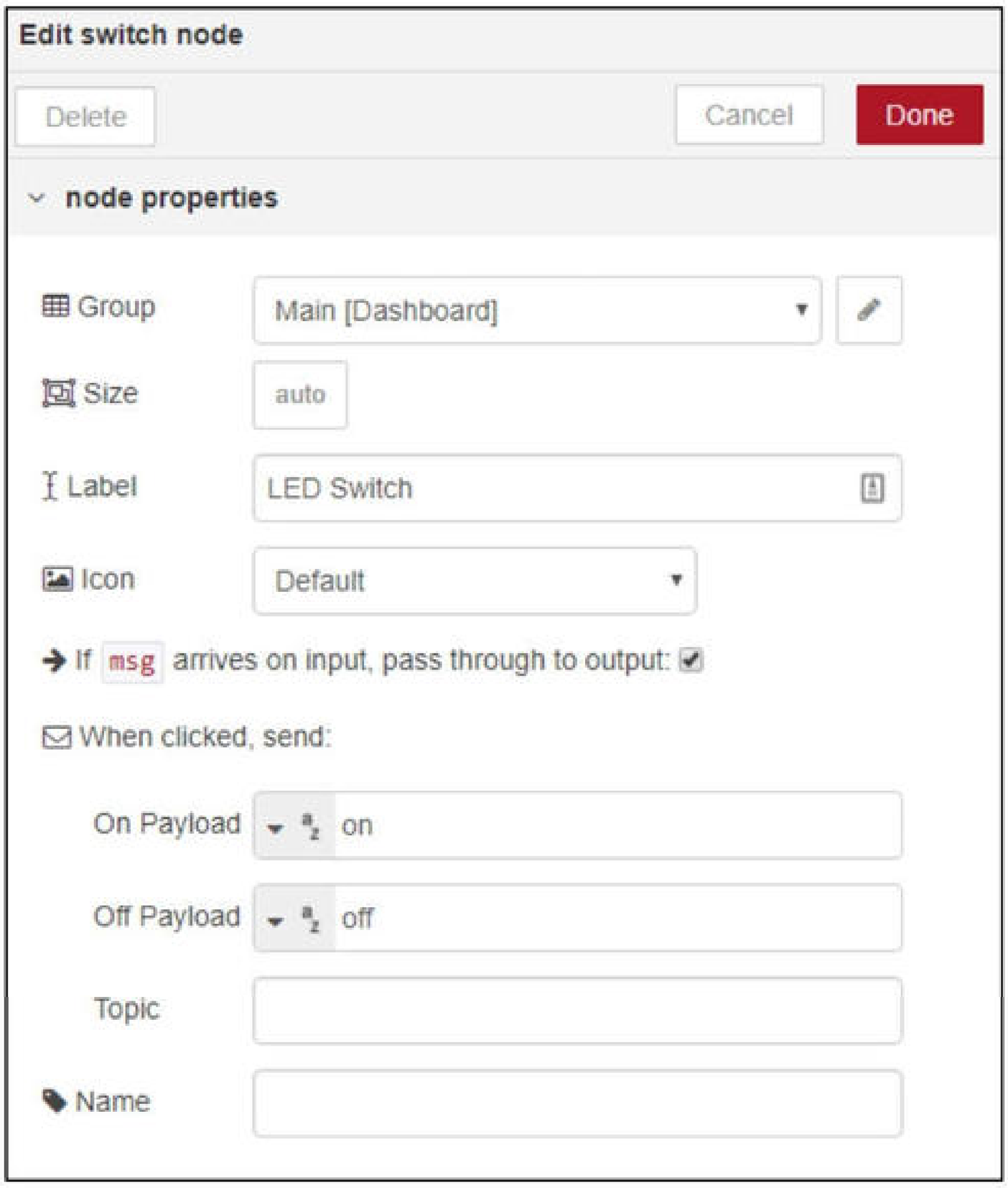


Understanding the flow

Let's look at each node and understand what they do.

LED Switch Node

The LED Switch node creates a switch in the UI (User Interface). The switch is connected to an MQTT node called "output". The switch sends different messages to the output accordingly to its state. If you double-click on the LED Switch node, you should get something as follows:

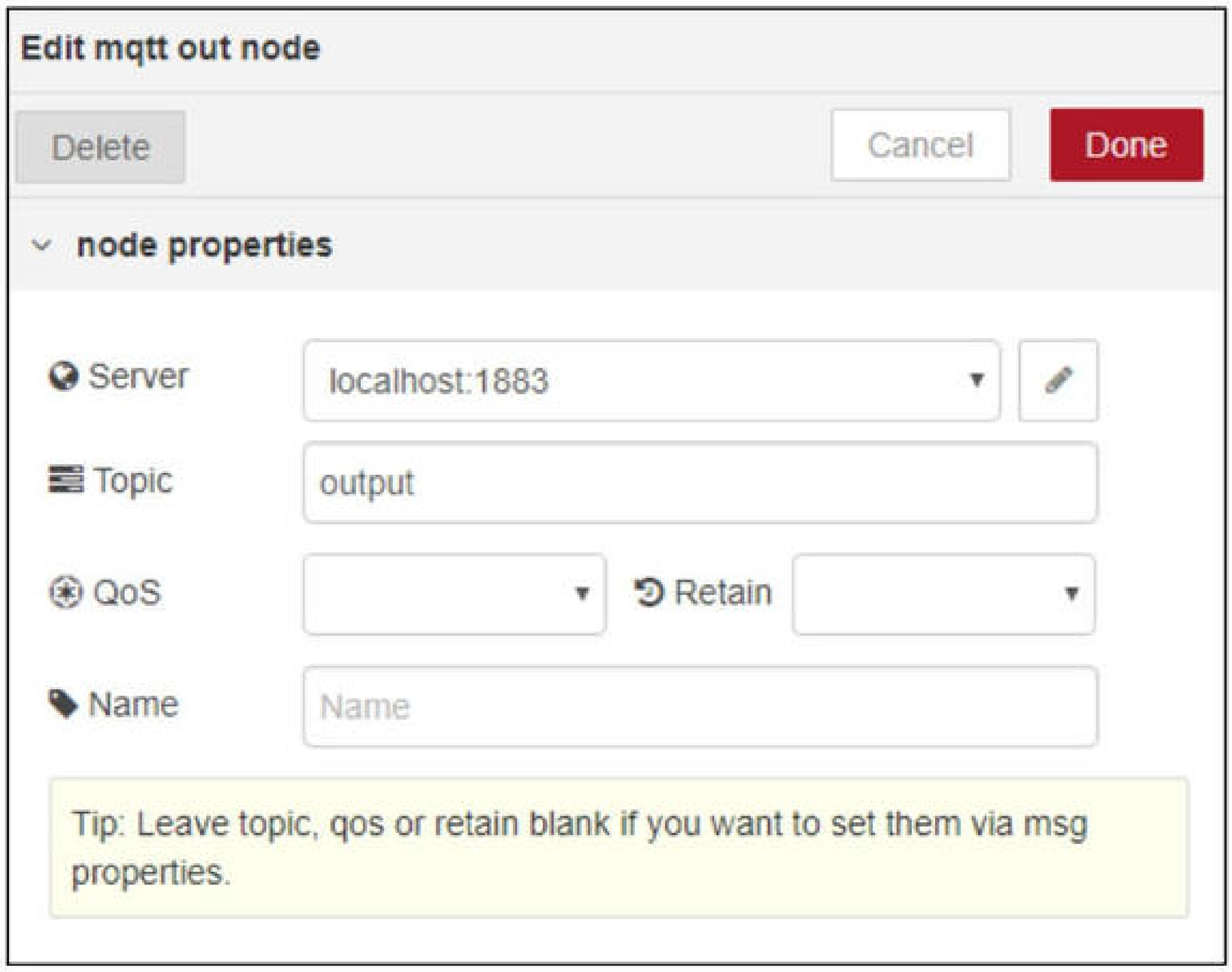


The "Group" field indicates in which part of the UI the switch appears. For this example, we'll leave the default settings.

When the switch is turned on ("On Payload"), we send a String message with "on" to the output (the MQTT output node). When the switch is off, we send a "off" message to the output.

MQTT output node

When the switch changes its state, it sends an "on" or "off" message to an output node (in this case an MQTTT output node). This node receives the messages and publishes them on a specific topic. Double-click the MQTT output node.

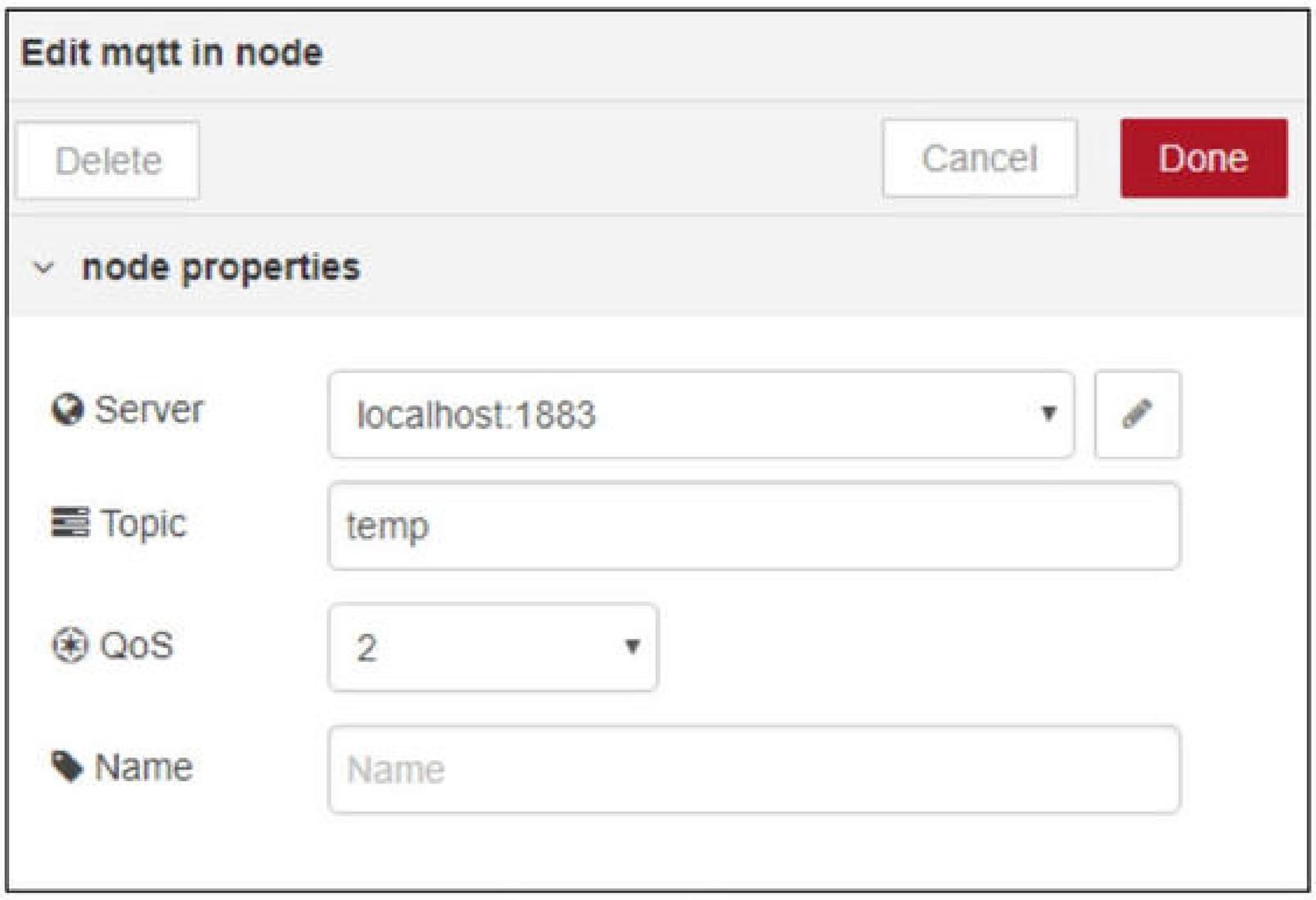


The important fields are the "Server" and the "Topic". The "Server" field refers to the MQTT broker. Because we're using the Raspberry Pi to run both Node-RED and Mosquitto broker, we can set the "Server" to "localhost".

The "topic" field refers to the topic where the messages will be published. As we've seen previously in this Unit, the switch publishes messages on the ***output*** topic.

MQTT temp node

The MQTT temp node listens for messages on a specific topic. In this case, Node-RED wants to receive messages on the ***temp*** topic. This is the topic the ESP32/ESP8266 is publishing the temperature readings. Double-click the MQTT temp node.



As with the previous node, we set the "Server" field to the "localhost". Set the topic to ***temp*** and the QoS to 2.

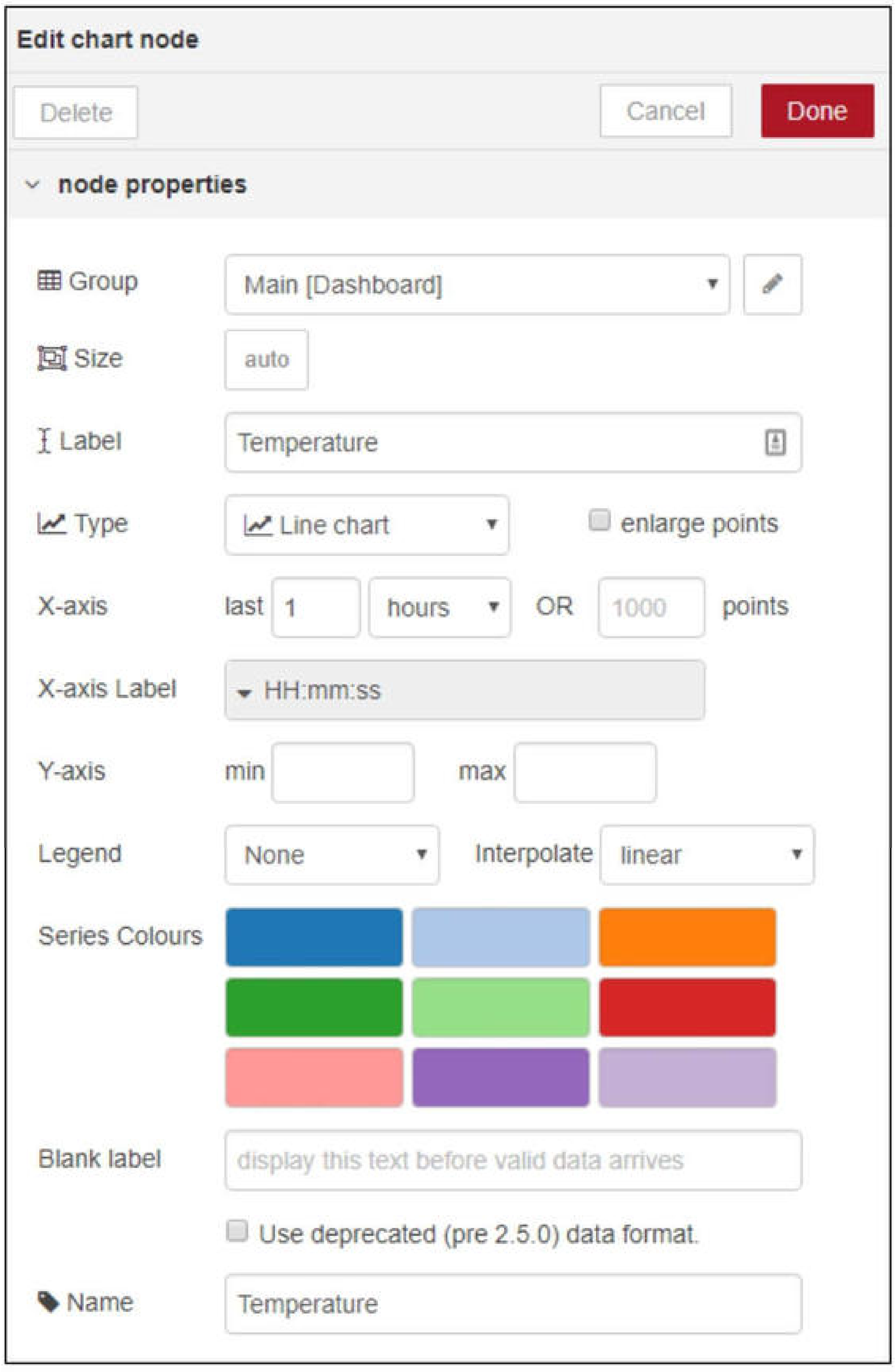
QoS means quality of service. A QoS of 2 ensures that the message is always delivered exactly once. You can learn more about MQTT QoS here.

In summary, the MQTT temp node receives the messages published on the ***temp*** topic and sends them to a chart.

Temperature chart node

The temperature chart node receives the temperature readings from the MQTT temp node and publishes them on a chart.

Double-click the temperature chart node. You can select the type of chart, the X-axis and the Y-axis label, color and others. In this example, the chart is set to display the sensor readings from the last hour.



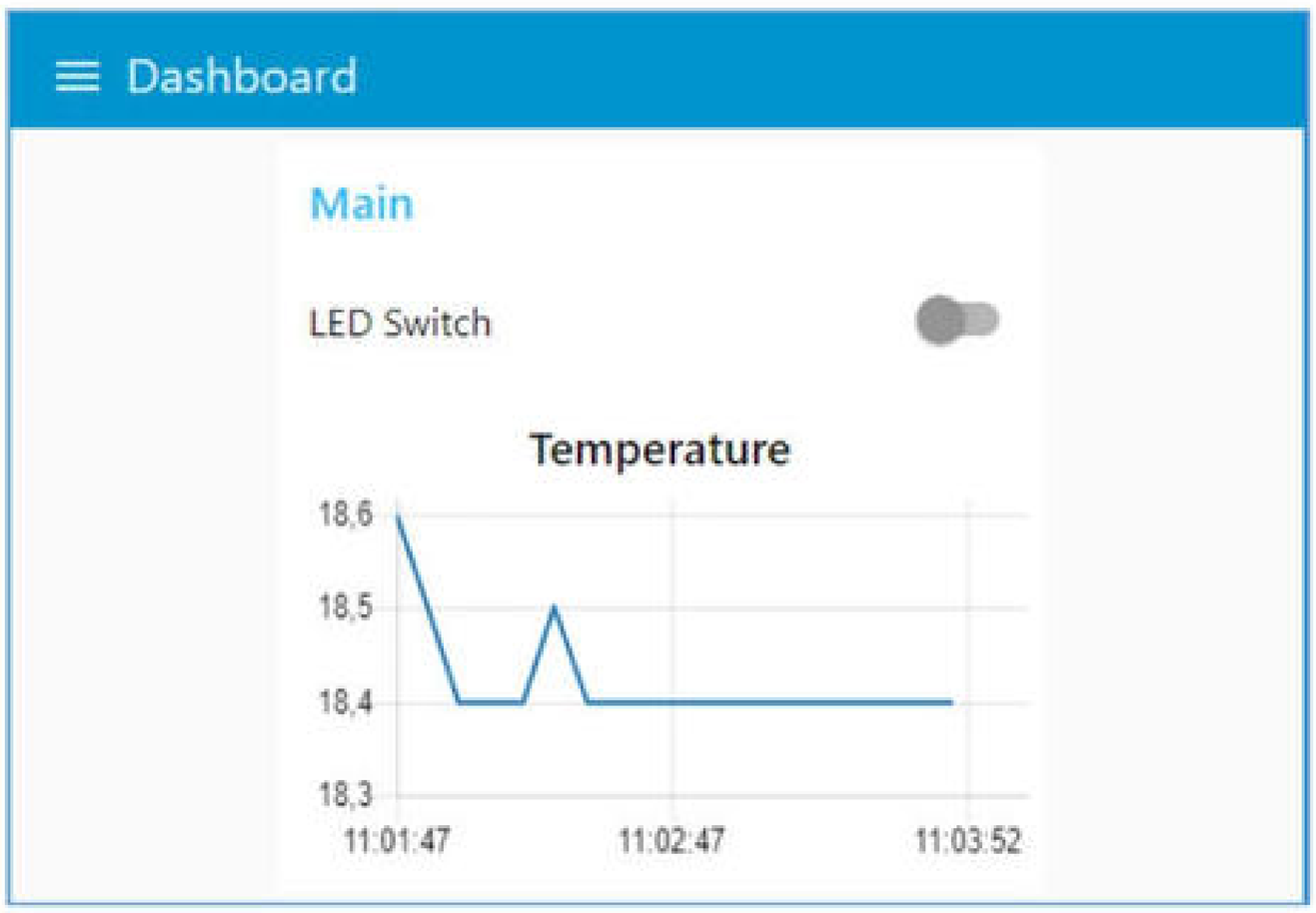
Node-RED

Click the **Deploy** button to save all the changes.

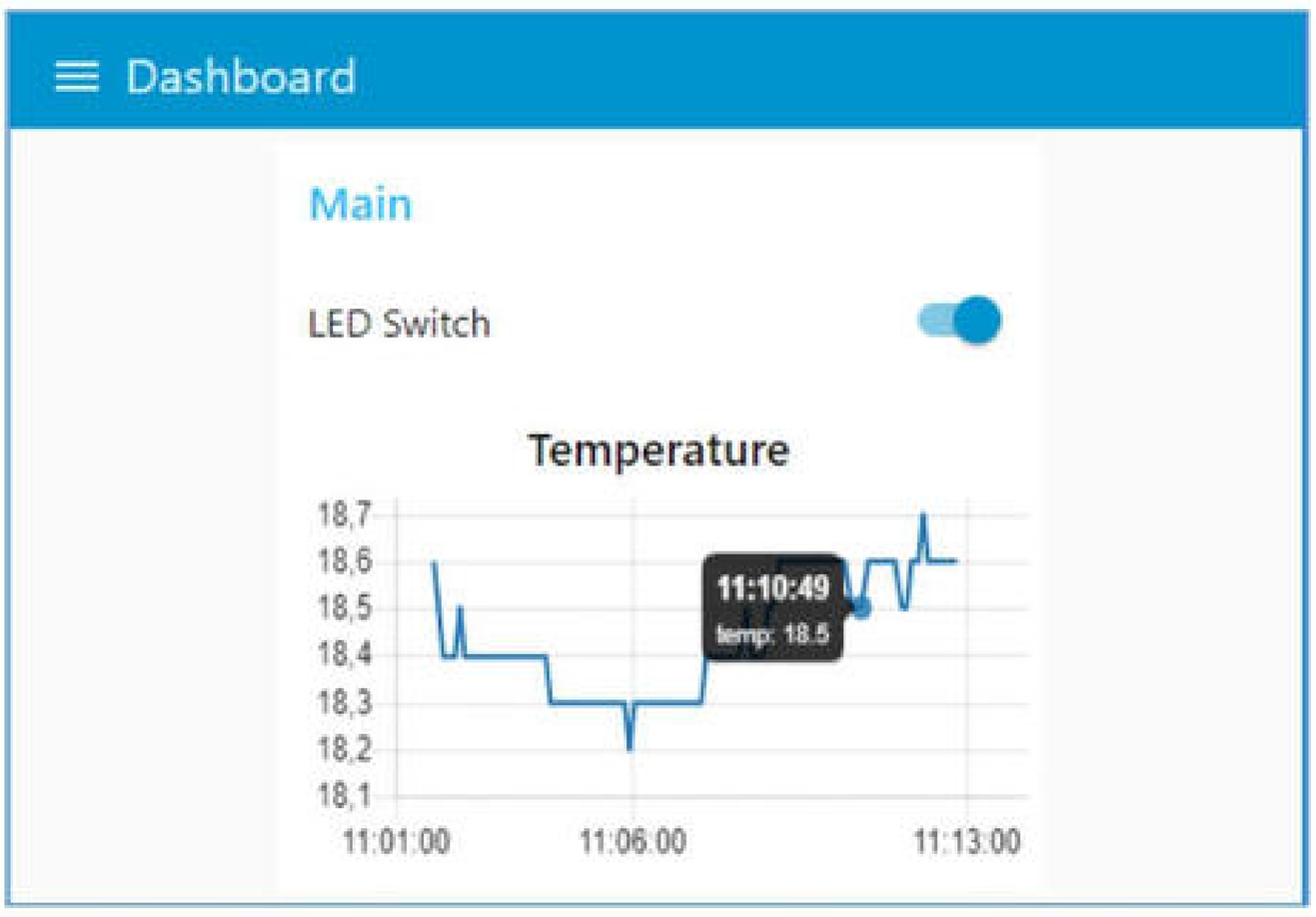
Now, your Node-RED application is ready. To access Node-RED Dashboard and see how your application looks, access any browser in your local network and type:

```
http://Your_RPi_IP_address:1880/ui
```

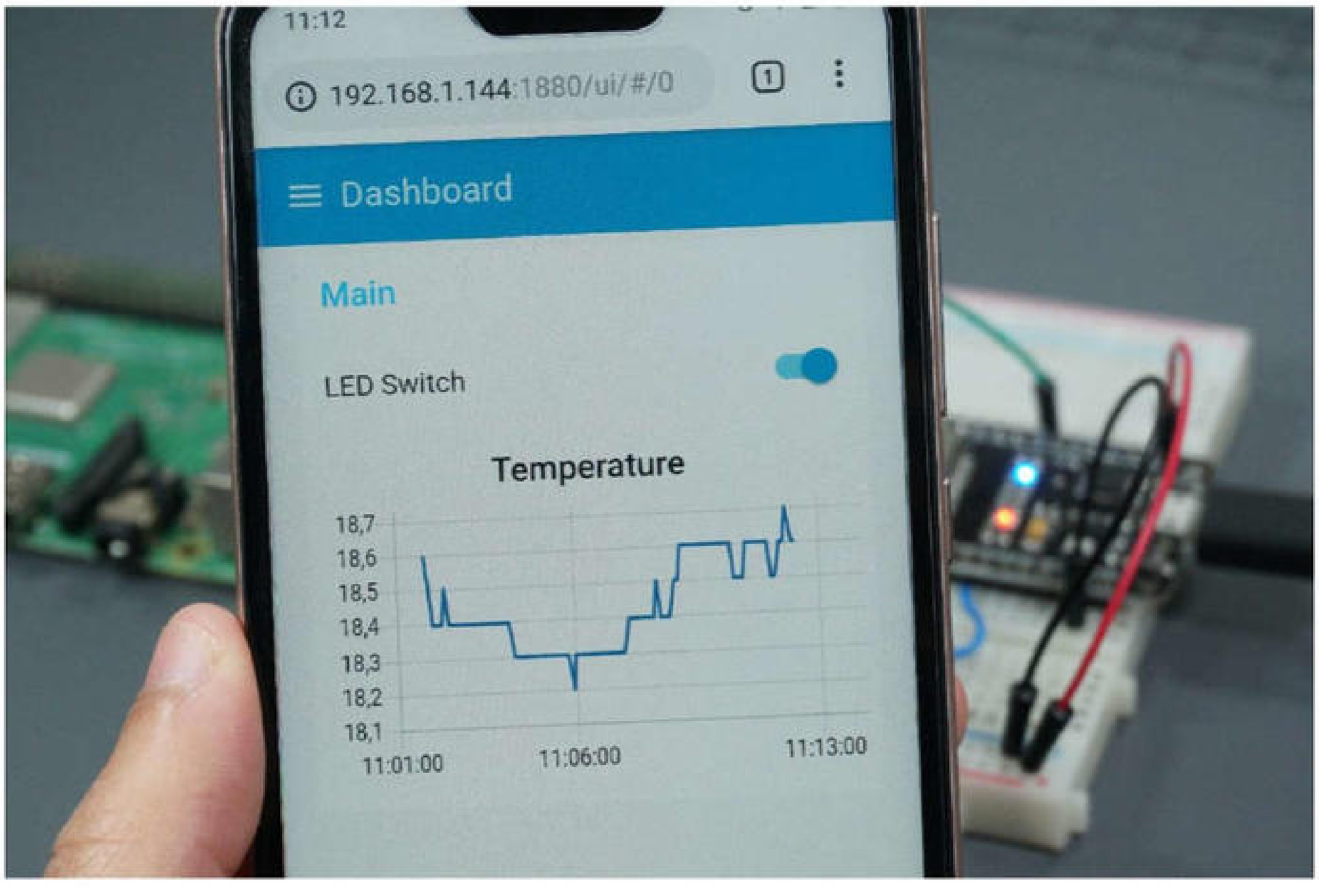
Your application should look as in the following figure. You should have an LED switch, and a Temperature chart.



You can check the data at a specific data point by sliding the cursor over a point in the chart.



Now, you should be able to control the ESP32/ESP8266 on-board LED from the Node-RED Dashboard and check the latest sensor readings on the chart.



Wrapping Up

In this project, you've learned the basic concepts to publish and subscribe messages using Node-RED and how to establish a communication between Node-RED and your ESP32/ESP8266 board.

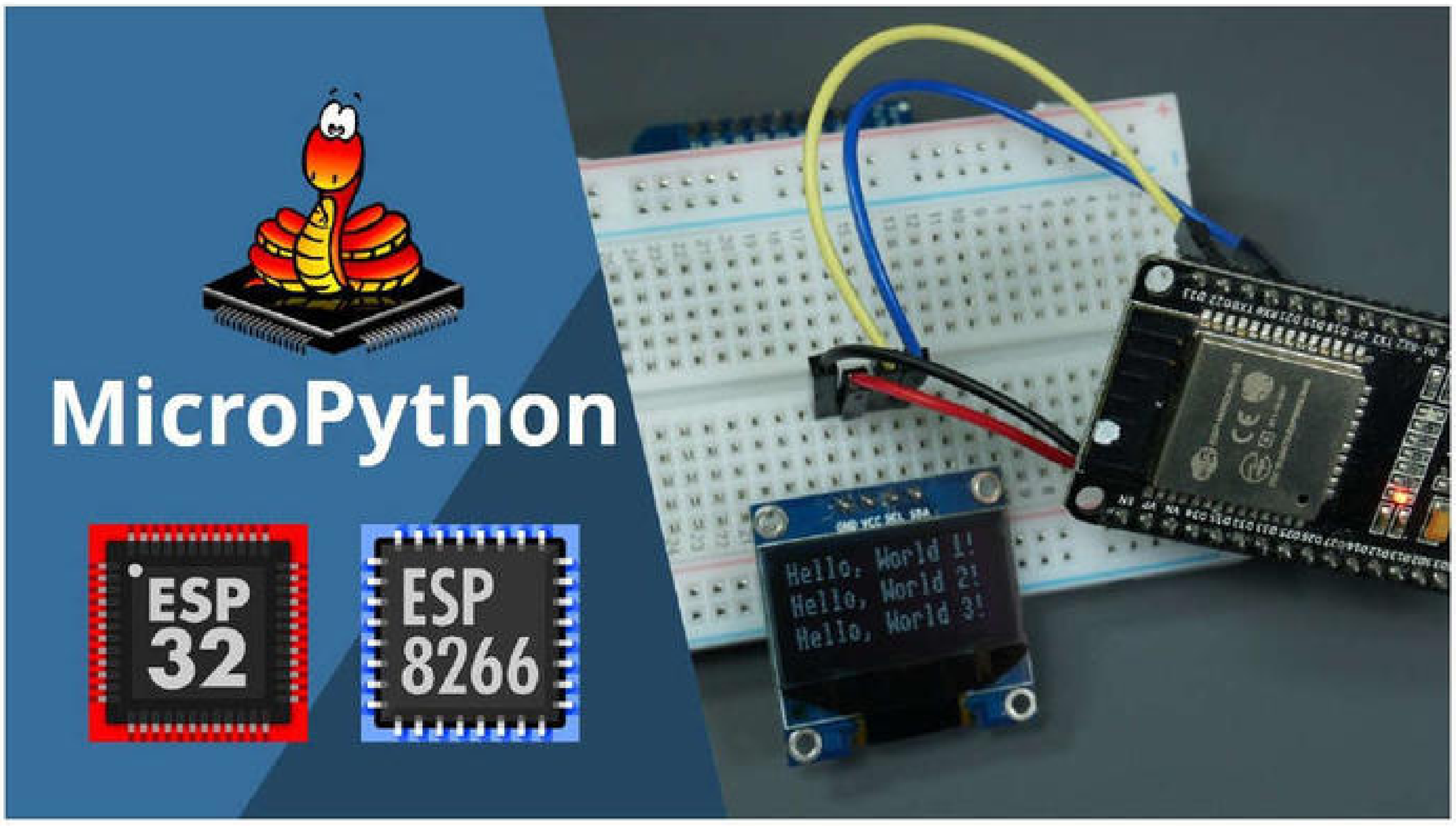
You can modify the project presented here to control any other outputs and display readings from other sensors or other pieces of information you may find useful in your home automation system.

MODULE 6

Other components:

OLED, DC Motor, Addressable RGB LED Strip

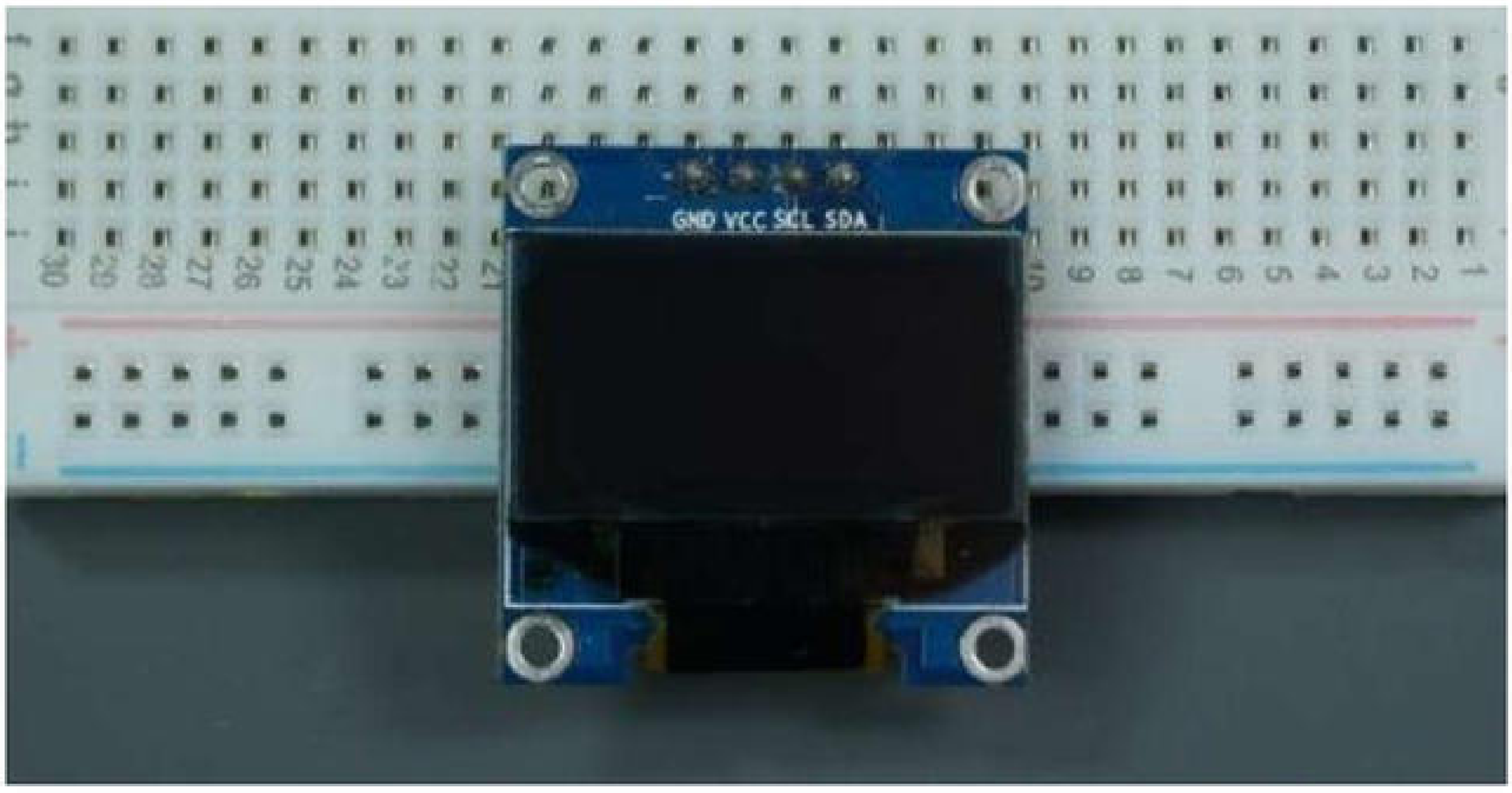
OLED Display



In this section, we'll show you how to use the 0.96 inch SSD1306 OLED display. As an example, we'll show you how to display a simple 'Hello World' message. Later, we'll also show you how to use other useful functions to interact with the OLED.

Introducing the OLED display

In this guide we'll use the 0.96 inch SSD1306 OLED display that is 128x64 pixels and uses I2C communication protocol.



I2C communication

For the I2C OLED display, these are the connections you need to make:

- **ESP32**: SCL: GPIO22; SDA: GPIO21
- **ESP8266**: SCL: GPIO5 (D1); SDA: GPIO4 (D2)

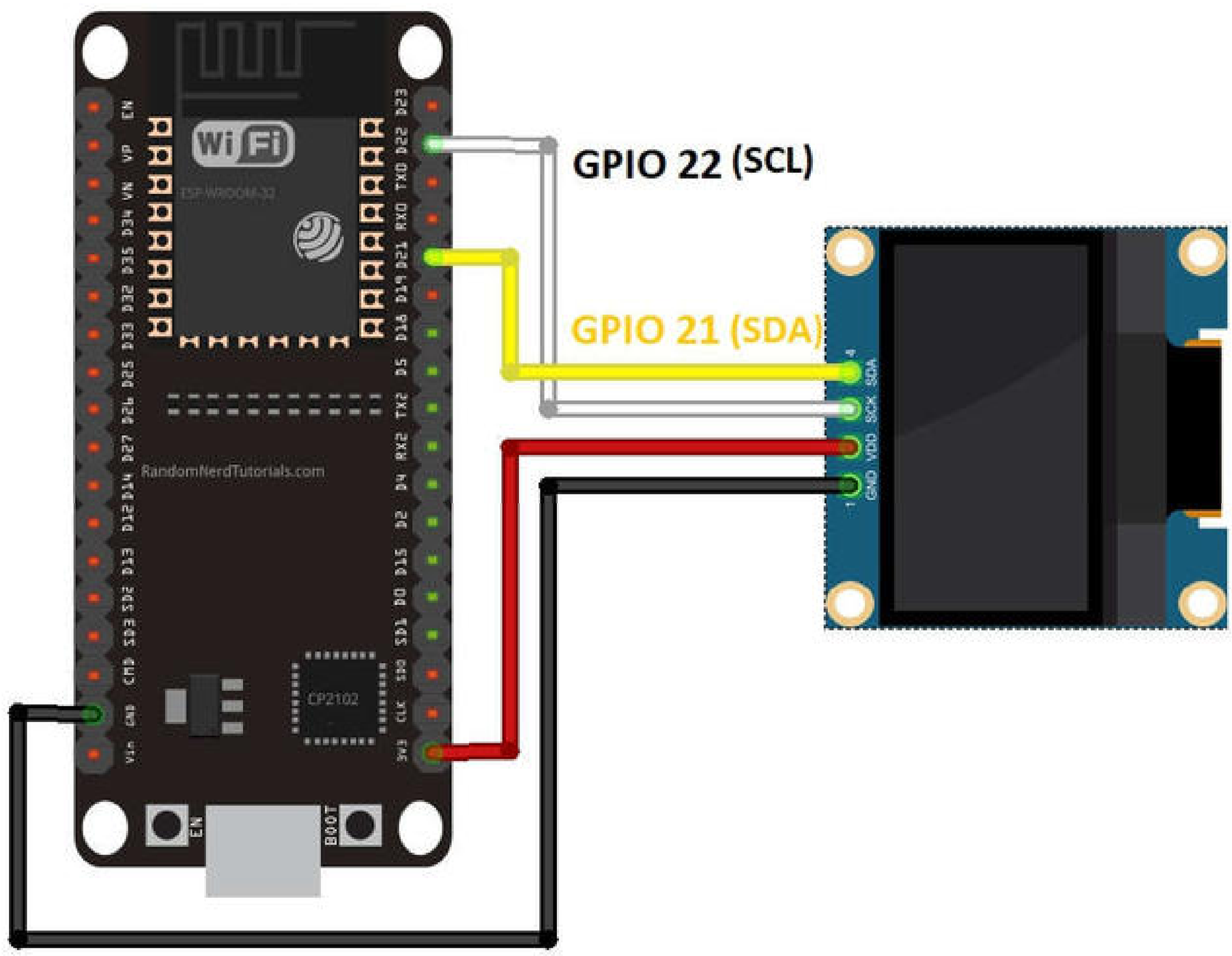
Parts Required

Here's a list of parts you need for this project:

- ESP32 or ESP8266
- SSD1306 128x64 - 0.96 inch OLED display
- Jumper wires

Schematic - ESP32

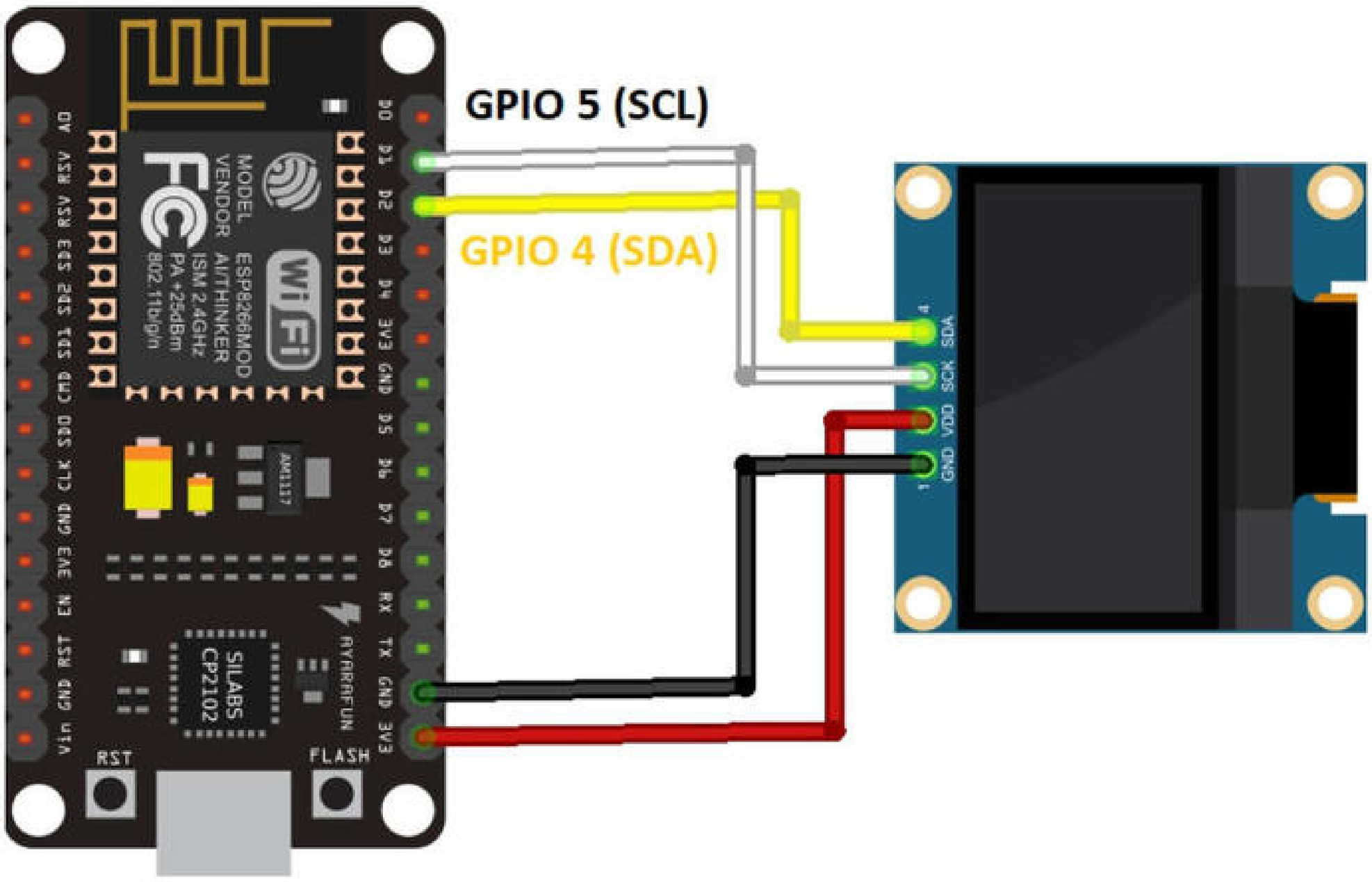
Follow the next schematic diagram if you're using an ESP32 board:



This schematic uses the ESP32 DEVKIT DOIT board version with 36 GPIOs. Before assembling the circuit double-check the pinout for the board you're using.

Schematic - ESP8266

Follow the next schematic diagram if you're using an ESP8266 board:



SSD1306 OLED Library

The library to write to the OLED display isn't part of the standard MicroPython library by default. So, you need to upload the library to your ESP32/ESP8266 board.

1. Create a new file by pressing the **New File** button.

2. Copy the OLED library code into it. The OLED library code can be found in the following link:

https://github.com/RuiSantosdotme/ESP-MicroPython/blob/master/code/Others/OLED/ssd1306.py

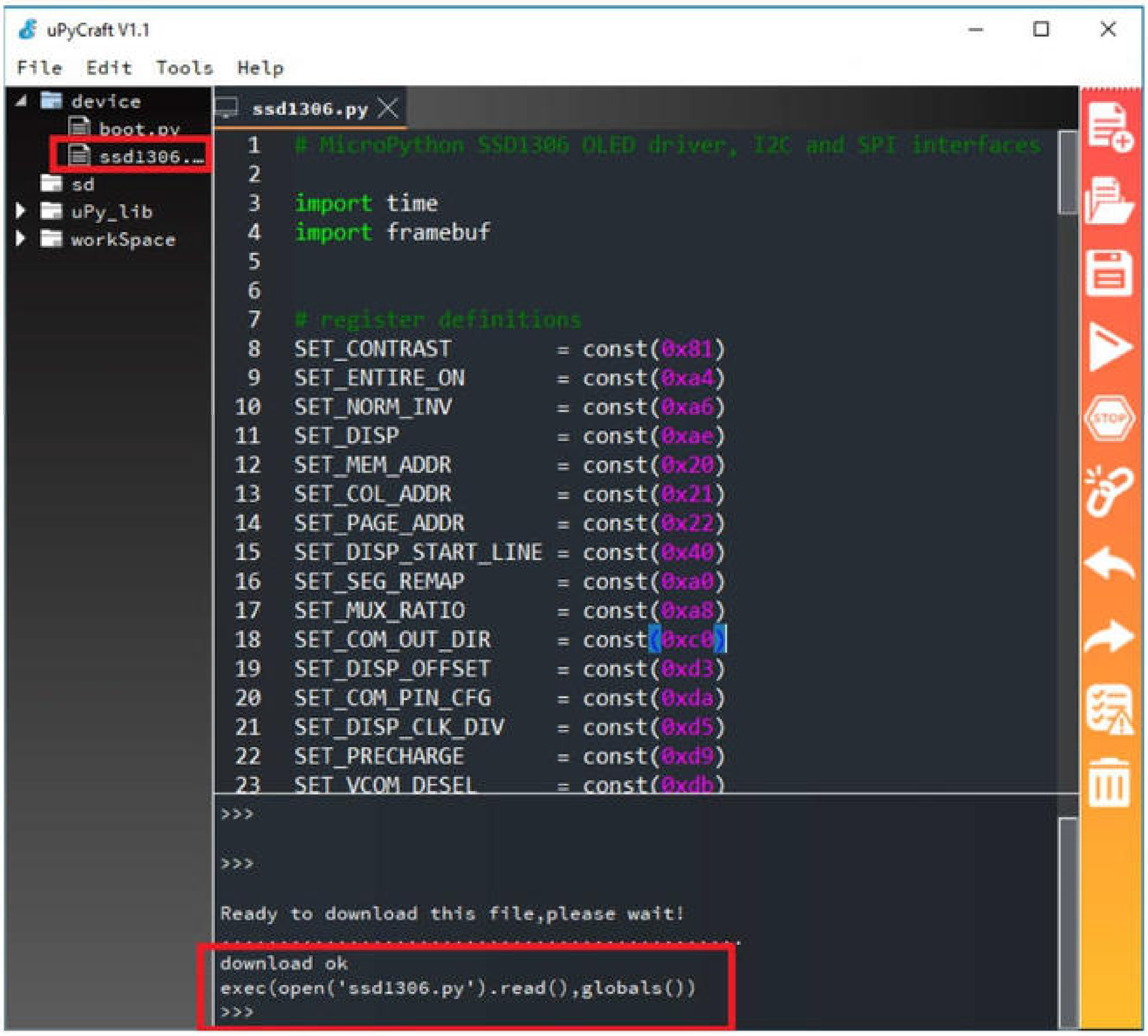
Note: the SSD1306 OLED display library was built by Adafruit and will no longer be updated. At the moment, it works fine. However, we'll update this guide if we find a similar library that works as well as this one.

3. Save the file by pressing the **Save** button.

4. Call this new file "*ssd1306.py*" and press ok.

5. Click the **Download and Run** button.

The file should be saved on the device folder with the name "ssd1306.*py*" as highlighted in the following figure.



Now, you can use the library functionalities in your code by importing the library.

Note: if you're using Thonny IDE, you just need to:

1. Copy the library code to a new file.

2. Name that file *ssd1306.py*.

3. Go to **Device ▸ Upload current script with the current name**.

Code

After uploading the library to the ESP32 or ESP8266, copy the following code to the *main.py* file. It simply prints the "Hello, World!" message three times in the display.

```
from machine import Pin, I2C
import ssd1306
from time import sleep

#ESP32 Pin assignment
i2c = I2C(-1, scl=Pin(22), sda=Pin(21))

# ESP8266 - Pin assignment
#i2c = I2C(-1, scl=Pin(5), sda=Pin(4))

oled_width = 128
oled_height = 64
oled = ssd1306.SSD1306_I2C(oled_width, oled_height, i2c)

oled.text('Hello, World 1!', 0, 0)
oled.text('Hello, World 2!', 0, 10)
oled.text('Hello, World 3!', 0, 20)

oled.show()
```

SOURCE CODE
https://github.com/RuiSantosdotme/ESP-MicroPython/blob/master/code/Others/OLED/main.py

How the Code Works

Start by importing the necessary modules to interact with the GPIOs and send data to the OLED via I2C communication. You need to import the `Pin` and `I2C` classes from the `machine` module.

```
from machine import Pin, I2C
```

You also need to import the OLED library that you previously uploaded to the board as *ssd1306.py* file.

```
import ssd1306
```

The ESP32 default I2C pins are GPIO 22 (SCL) and GPIO 21 (SDA). The ESP8266 default I2C pins are GPIO 5 (SLC) and GPIO 4 (SDA).

Use the following line if you're using an ESP32 board:

```
#ESP32 Pin assignment
i2c = I2C(-1, scl=Pin(22), sda=Pin(21))
```

Comment the previous line and uncomment the following if you're using an ESP8266 board:

```
#ESP8266 Pin assignment
i2c = I2C(-1, scl=Pin(5), sda=Pin(4))
```

Define the OLED width and height on the following variables:

```
oled_width = 128
oled_height = 64
```

After that, create an `SSD1306_I2C` object called `oled`. This object accepts the OLED width, height, and the I2C pins you've defined earlier.

```
oled = ssd1306.SSD1306_I2C(oled_width, oled_height, i2c)
```

After initializing the OLED display, you just need to use the `text()` function on the `oled` object to write text. After the `text()` function, you need to call the `show()` method to update the OLED.

```
oled.text('Hello, World 1!', 0, 0)
oled.text('Hello, World 2!', 0, 10)
oled.text('Hello, World 3!', 0, 20)

oled.show()
```

The `text()` method accepts the following arguments (in this order):

- Message: must be of type String
- X position: where the text starts
- Y position: where the text is displayed vertically
- Text color: it can be either black or white. The default color is white and this parameter is optional.
    - 0 = black
    - 1 = white

For example, the following line writes the '`Hello, World 1`!' message in white color. The text starts on x = 0 and y = 0.

```
oled.text('Hello, World 1!', 0, 0)
```

The next line of code writes the text on the next line (y =10).

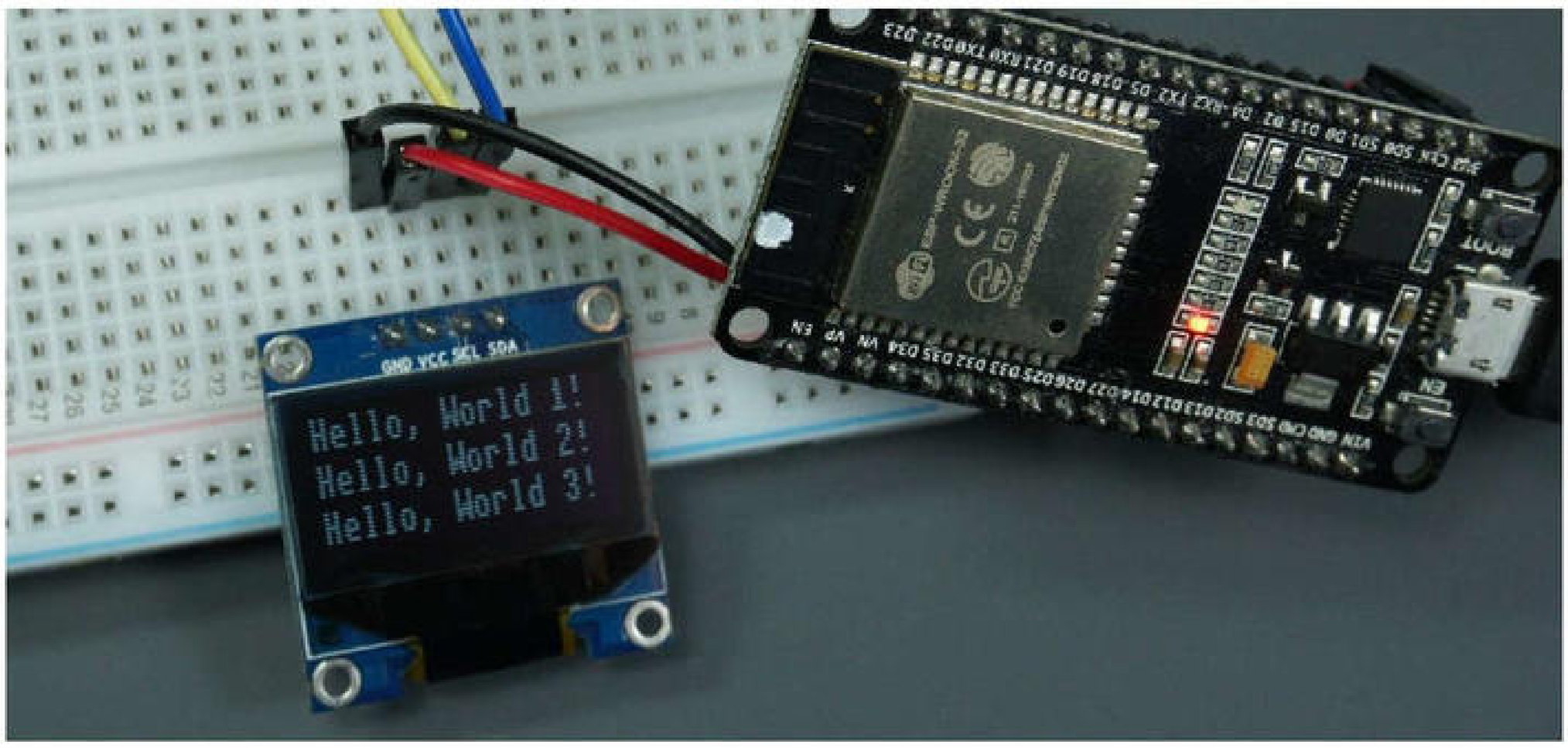
```
oled.text('Hello, World 2!', 0, 10)
```

Finally, for the changes to take effect, use the `show()` method on the `oled` object.

```
oled.show()
```

Demonstration

Upload the code to your board. Your OLED display should look as follows:



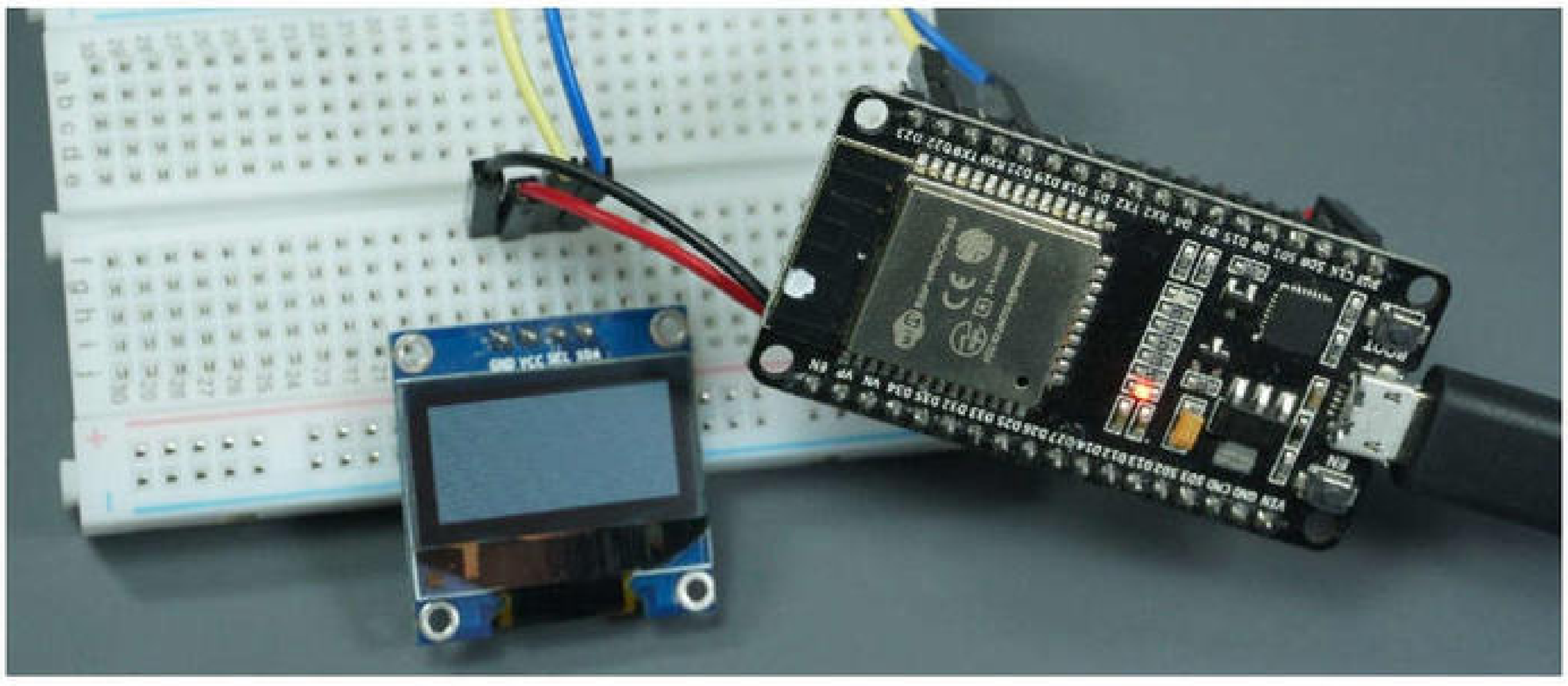
Other OLED functions

There are other methods you can use on the `oled` object to perform other tasks.

For example:

- To fill the entire screen with white, use the `fill()` function as follows:

```
oled.fill(1)
oled.show()
```



- Clear the screen (all black):

```
oled.fill(0)
oled.show()
```

- Draw a pixel:

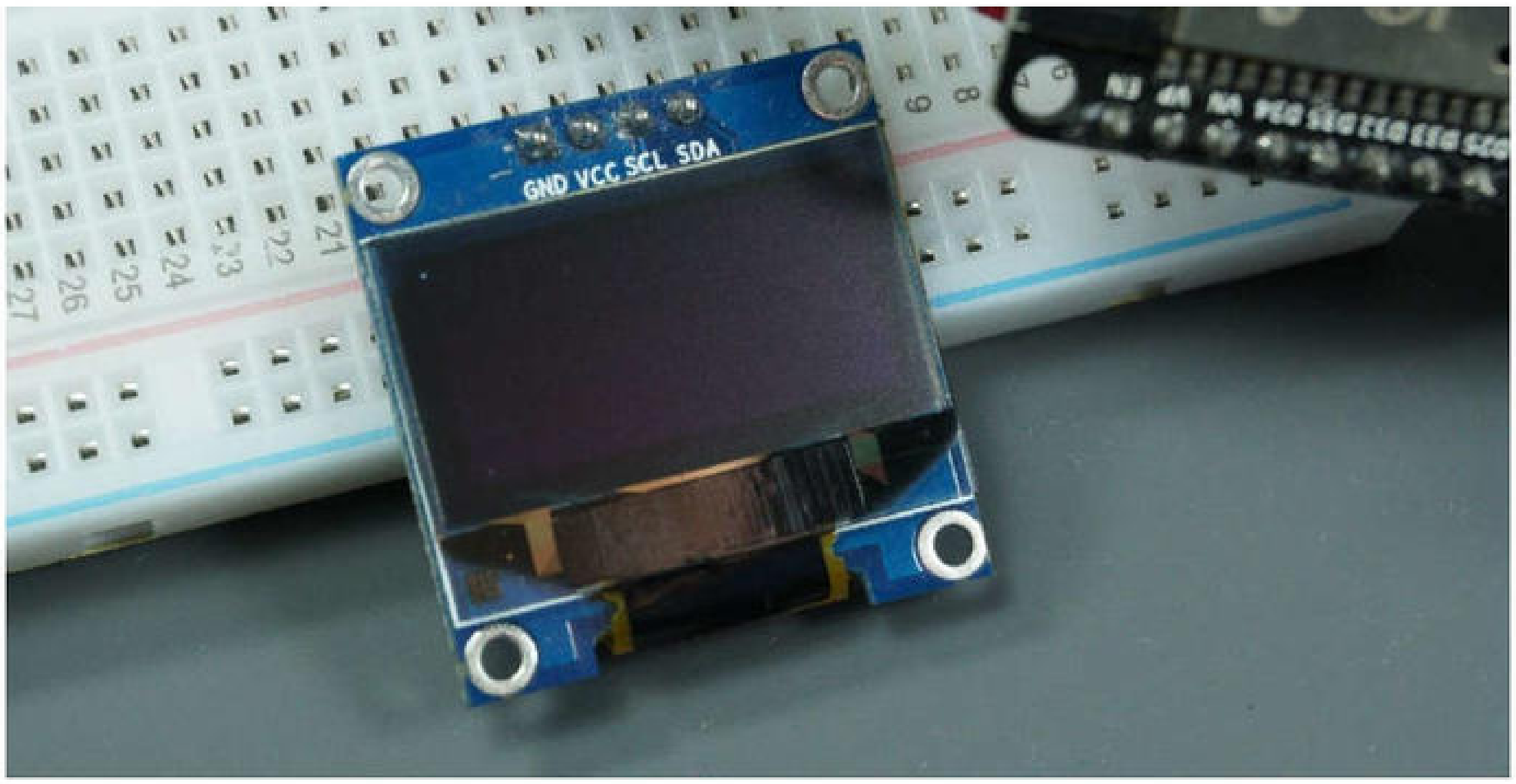
To draw a pixel, you use the `pixel()` method, followed by the `show()` method.

The `pixel()` method accepts the following arguments:

- X coordinate: pixel location horizontally
- Y coordinate: pixel location vertically
- Pixel color:
    - 0 = black
    - 1 = white

For example, to draw a white pixel on the top left corner:

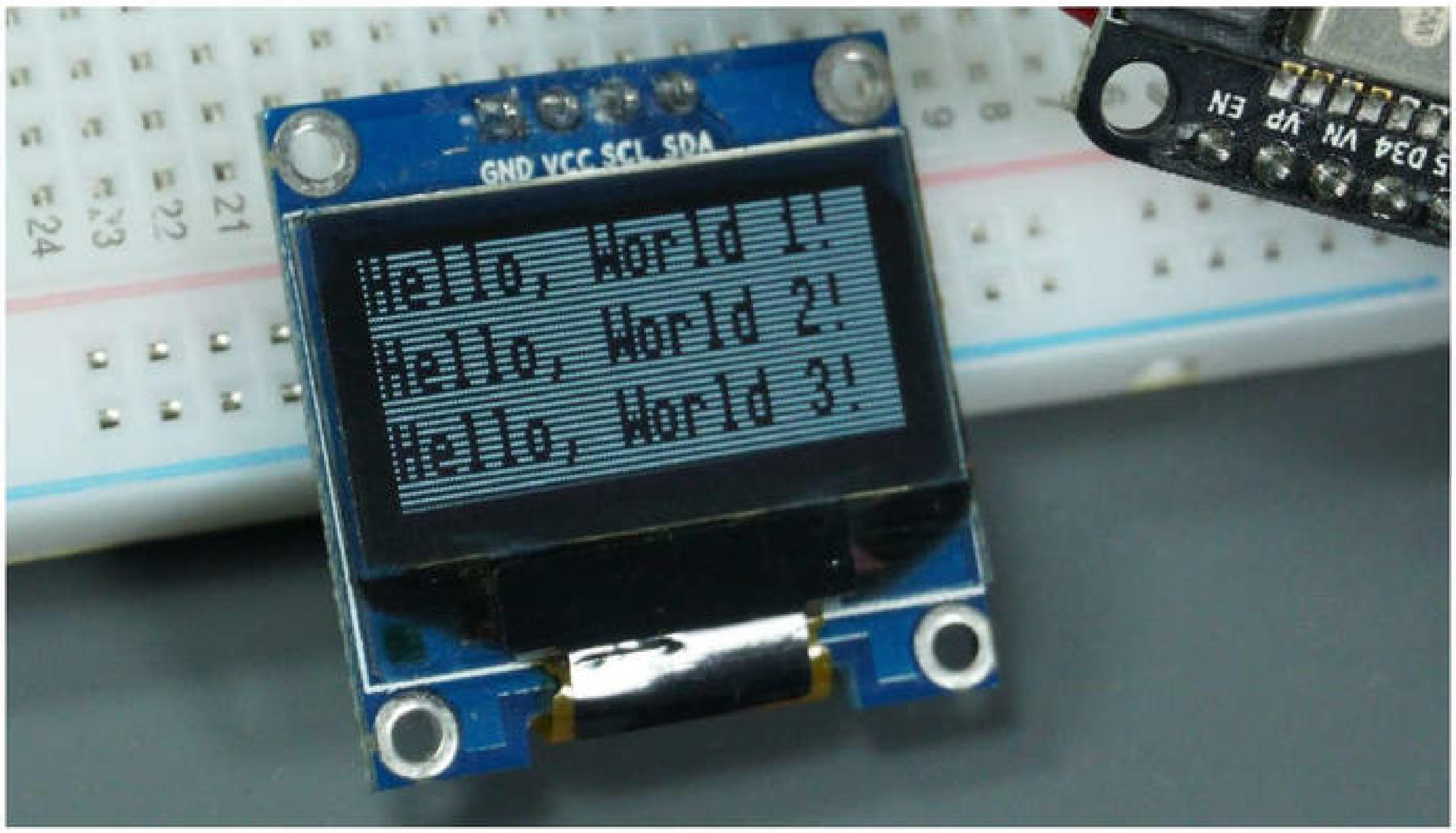
```
oled.pixel(0, 0, 1)
oled.show()
```



- Invert black/white:

You can also invert the white with black colors and vice versa, using the `invert()` method:

```
oled.invert(True)
```



To get back to the original colors, use:

```
oled.invert(False)
```

Displaying data from sensors

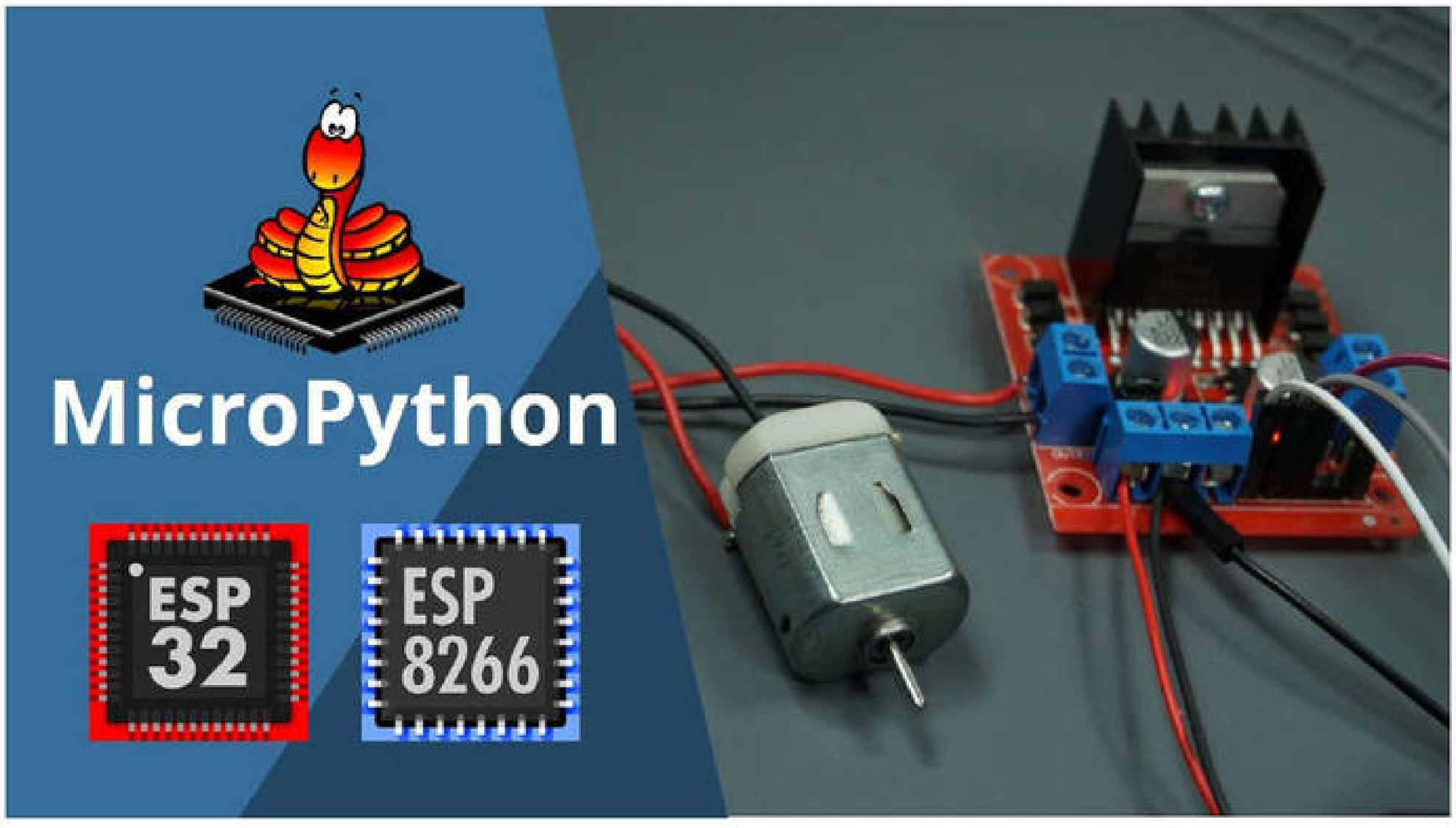
The `text()` function only accepts variables of type *String* as a message. Sensor readings are usually stored in *int* or *float* variables. If you want to display sensor readings and they are stored in *int* or *float* variables, they should be converted to a String. To convert the data to a string you can use the *str()* function:

```
temperature = 12.34
temperature_string = str(temperature)
```

Wrapping Up

This quick guide showed you how to use the OLED basic functionalities: write text and draw pixels. Now, you can use the OLED to display messages, or sensor readings.

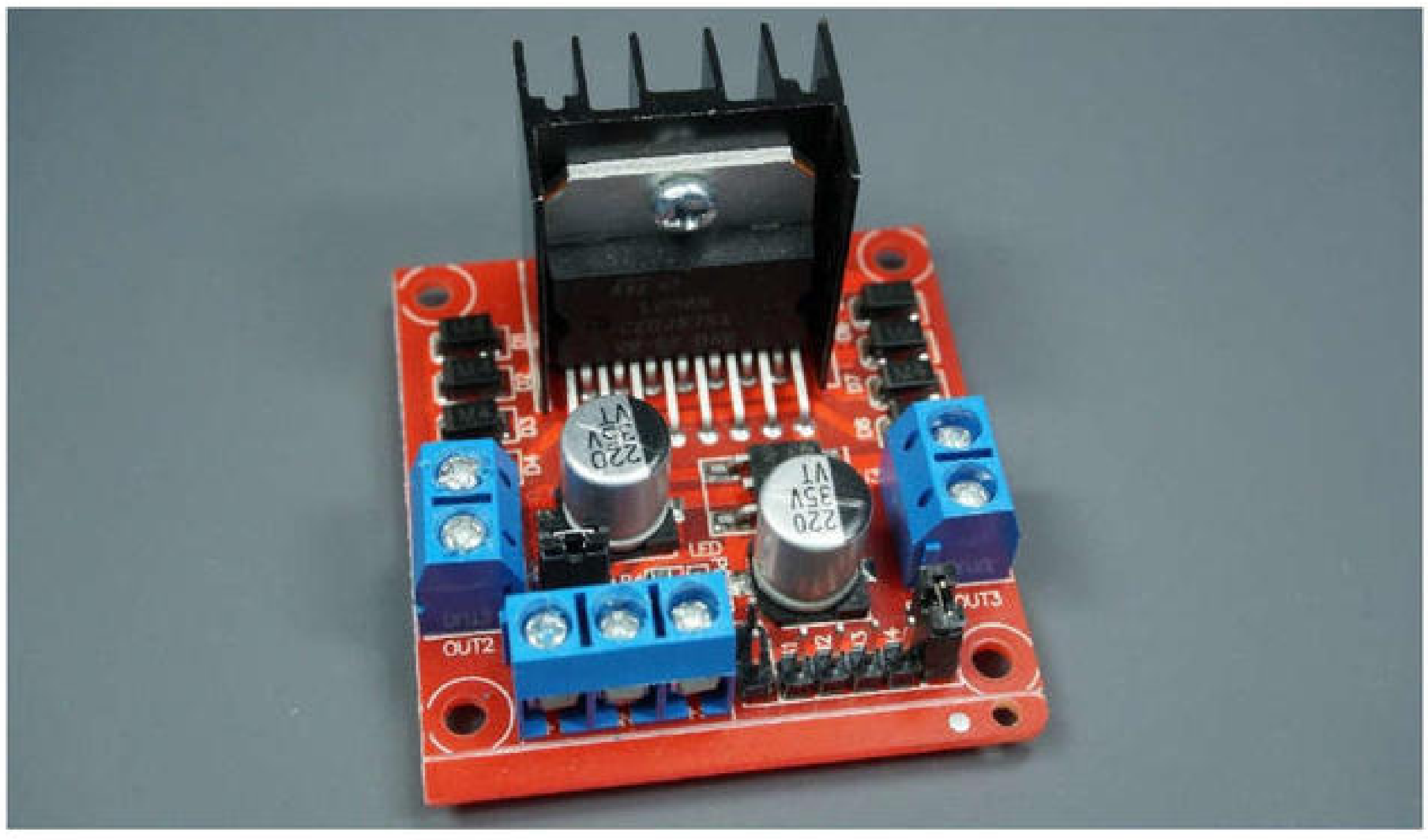
Control a DC Motor



This section shows how to control a DC motor with your ESP32 or ESP8266 using MicroPython. You'll learn how to make it go forward, backwards, and stop by building a Python module.

Introducing the L298N Motor Driver

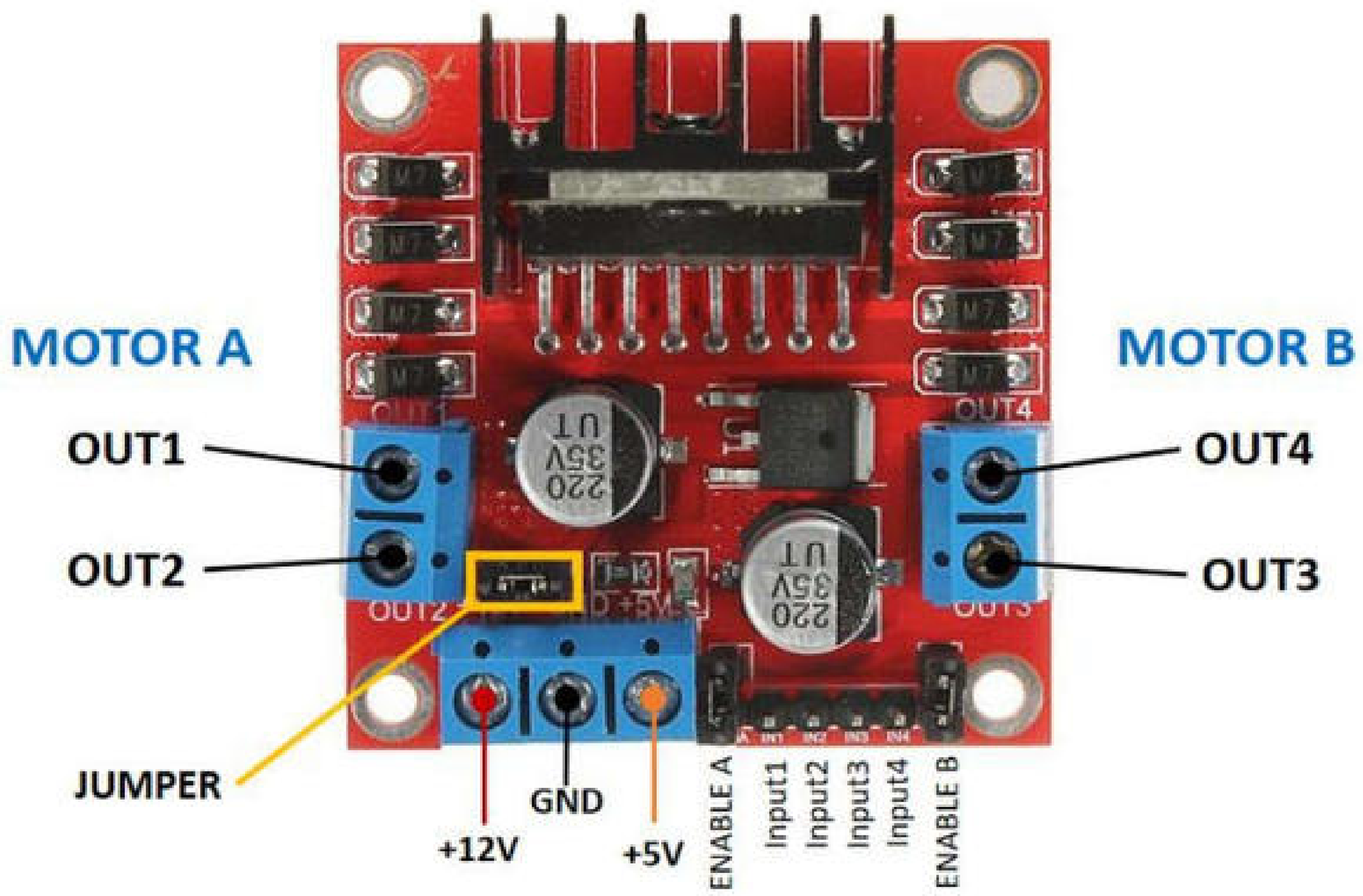
There are many ways to control DC motors. We'll use the L298N motor driver, but any similar motor driver should work.



Using the L298N motor driver is suitable for most hobbyist motors that require 6V to 12V to operate. Additionally, it allows you to drive two DC motors simultaneously, which is perfect if you want to build a robot.

L298N Motor Driver Pinout

Let's look at the L298N motor driver pinout and see how it works.



The motor driver has a two-terminal block in each side for each motor. OUT1 and OUT2 at the left and OUT3 and OUT4 at the right.

- **OUT1**: DC motor A + terminal
- **OUT2**: DC motor A – terminal
- **OUT3**: DC motor B + terminal
- **OUT4**: DC motor B – terminal

At the bottom you have a three-terminal block with **+12V**, **GND**, and **+5V**. The **+12V** terminal block is used to power up the motors. The **+5V** terminal is used to power up the L298N chip. However, if the jumper is in place, the chip is powered using the motor's power supply. So, in this case you don't supply 5V through the **+5V** terminal.

Note: if you supply more than 12V, you need to remove the jumper and supply 5V to the +5V terminal.

Important: despite the +12V terminal name, you can supply any voltage between 6V and 12V.

In summary:

- **+12V**: this terminal is where you should connect your power supply (input voltage is between 6V and 12V).
- **GND**: GND power supply.
- **+5V**: provide 5V if jumper is removed. Acts as a 5V output if jumper is in place.
- **Jumper**: jumper in place – uses the motors power supply to power up the chip. Jumper removed: you need to provide 5V to the +5V terminal. If you supply more than 12V, you should remove the jumper.

At the bottom right you have four input pins and two enable pins. The input pins are used to control the direction of your DC motors, and the enable pins are used to control the speed of each motor.

- **IN1** – Input 1 for Motor A
- **IN2** – Input 2 for Motor A
- **IN3** – Input 1 for Motor B
- **IN4** – Input 2 for Motor B
- **EN1** – Enable pin for Motor A
- **EN2** – Enable pin for Motor B

There are jumper caps on the enable pins by default. You need to remove those jumper caps to be able to control the speed of your motors.

Control DC motors with the L298N

Now that you're familiar with the L298N motor driver, let's see how to use it to control your DC motors.

The Enable Pins

The enable pins are like an ON and OFF switch for your motors. For example:

- If you send a **HIGH** (1) signal to the enable 1 pin, motor A is ready to be controlled and at the maximum speed.
- If you send a **LOW** (0) signal to the enable 1 pin, motor A turns off.
- If you send a **PWM** signal, you can control the speed of the motor. The motor speed is proportional to the duty cycle. However, note that for small duty cycles, the motors might not spin, and make a continuous buzz sound. It also depends on the frequency used.

Signal sent to the enable pin	Motor state
HIGH (1)	Motor enabled
LOW (0)	Motor not enabled
PWM	Motor enabled: speed proportional to duty cycle

The Input Pins

The input pins control the direction the motors are spinning. Input 1 and input 2 control motor A, and input 3 and 4 control motor B.

If you apply a LOW signal to input 1 and a HIGH signal to input 2, motor A will spin forward. If you apply power the other way around: HIGH to input 1 and LOW to input 2, the motor will rotate backwards. Sending LOW to input 1 and input 2 stops the motor. Motor B is controlled using the same method.

Direction	Input 1	Input 2	Enable 1
Forward	0	1	1
Backwards	1	0	1
Stop	0	0	0

Tip (*if you want to build a robot*) for your robot to move forward, both motors should be rotating forward. To make it go backwards, both should be rotating backwards.

To turn the robot in one direction, you need to spin the opposite motor faster. For example, to make the robot turn right, enable the motor at the left, and disable the motor at the right.

Now that you know how to control DC motors with the L298N motor driver, let's assemble the circuit and then, write the code.

Wiring the Circuit

For this example, you need to wire one DC motor to the ESP8266 or ESP32 board through the L298N motor driver. It's advisable to power up the motor using an independent power supply. Here we'll be using 4 AA 1.5V batteries that combined output approximately 6V.

Parts Required

Here's a list of parts you need to build the circuit:

- ESP32 or ESP8266
- L298N motor driver
- Mini DC Motor
- 0.1 µF ceramic capacitor
- Jumper wires

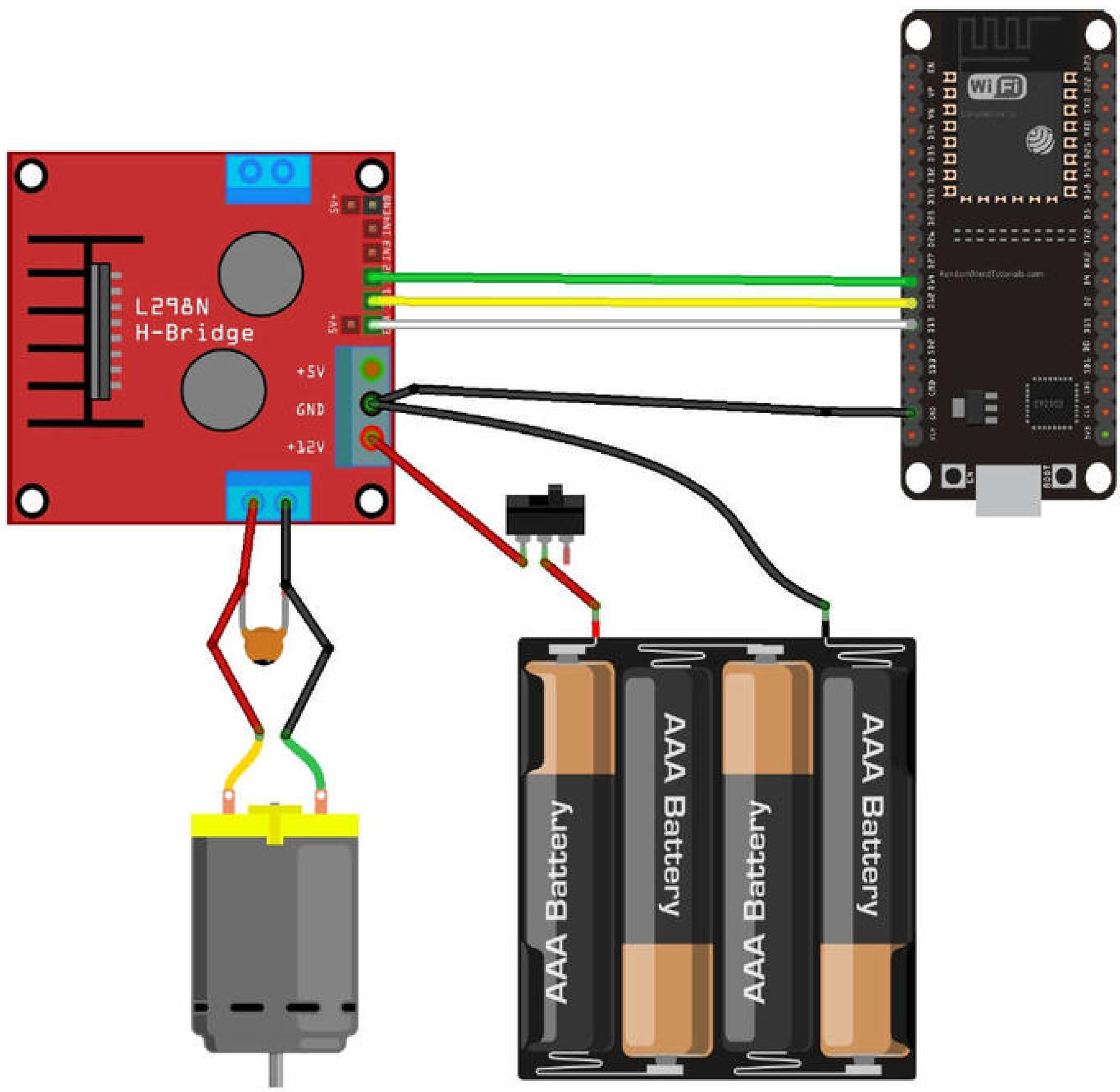
Wiring

The following table shows the connections between the ESP32/ESP8266 and the L298N Motor Driver.

L298N Motor Driver	Input 1	Input 2	Enable
ESP32	GPIO12	GPIO14	GPIO13
ESP8266	GPIO12 (D6)	GPIO14 (D5)	GPIO13 (D7)

Schematic – ESP32

Follow the next schematic diagram if you're using an ESP32 board:



This schematic uses the ESP32 DEVKIT DOIT board version with 36 GPIOs. Before assembling the circuit double-check the pinout for the board you're using.

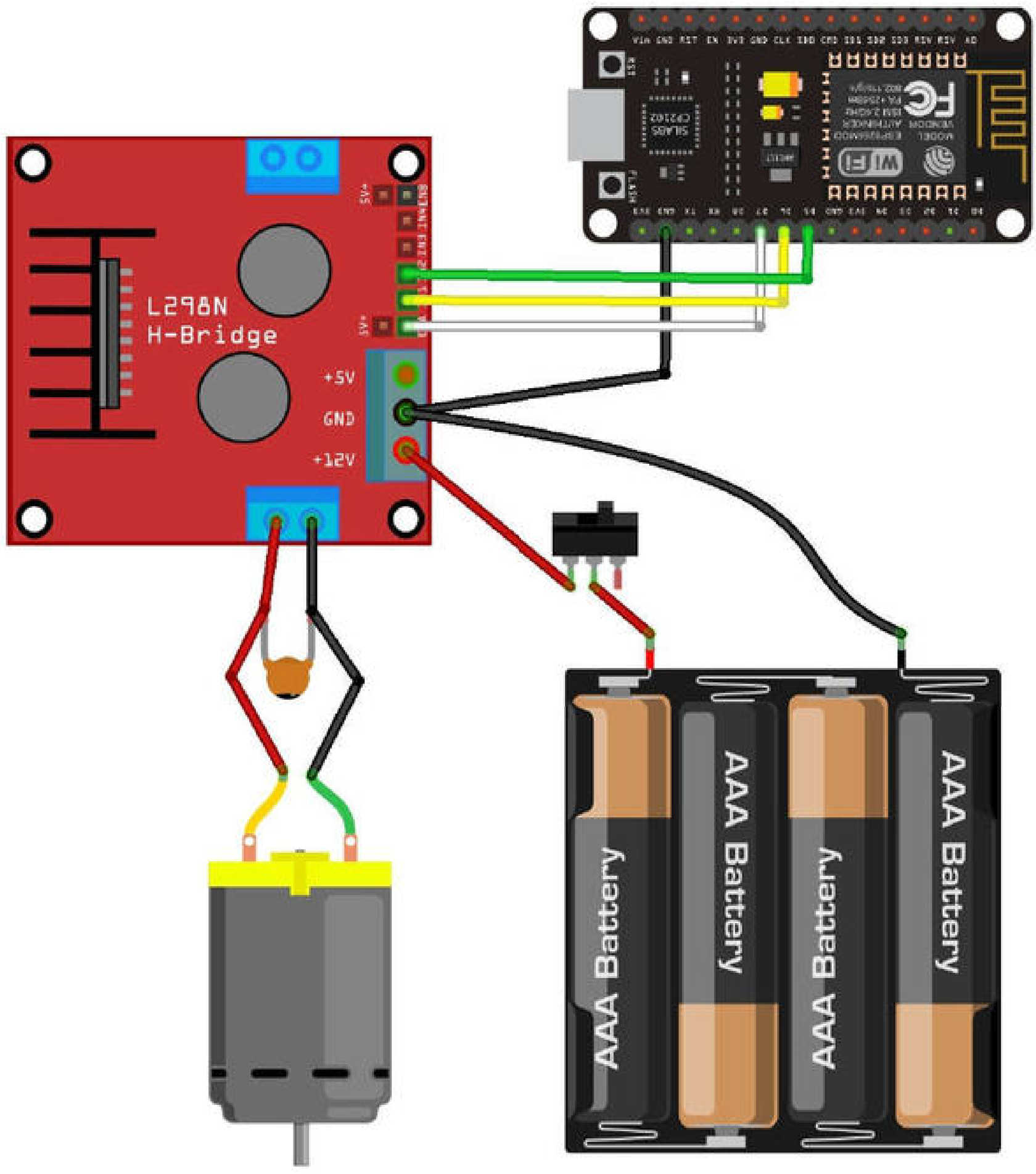
Note: we recommend soldering a 0.1 µF ceramic capacitor to the positive and negative terminals of each motor, as shown in the diagram to help smooth out any voltage spikes.

Additionally, you can solder a slider switch to the red wire that comes from the battery pack. This way, you can turn the power that goes to the motors and motor driver on and off.

The ESP32 should be powered using another power supply (for example, a power bank).

Schematic – ESP8266

Follow the next schematic diagram if you're using an ESP8266 board:



Note: we recommend soldering a 0.1 µF ceramic capacitor to the positive and negative terminals of each motor, as shown in the diagram to help smooth out any voltage spikes.

Additionally, you can solder a slider switch to the red wire that comes from the battery pack. This way, you can turn the power that goes to the motors and motor driver on and off.

The ESP8266 should be powered using another power supply (for example, a power bank).

Code

Writing the code to control the DC motor is not difficult, but it requires some repetitive commands. It would be nice to have a module (library) that would allow us to control the motor speed and direction with a single command.

With that in mind, we'll create a simple Python module with the basic commands to initialize a motor and control its speed and direction.

Creating the *dcmotor* Module

Before writing the code for this module, it's important to write down its functionalities:

- The library will be called *dcmotor.py* and it will have a single class called `DCMotor` with several methods.
- We should be able to initialize the motor using the input pin 1, input pin 2, and the enable pin as follows:

```
motor1 = DCMotor(Pin1, Pin2, enable)
```

`Pin1` and `Pin2` should be initialized as output pins, and the enable should be initialized as a PWM pin.

- To move the motor forward, we want to use a method called `forward()` on the `DCMotor` object that accepts the speed as a parameter. For example:

```
motor1.forward(speed)
```

The `speed` should be an integer number between 0 and 100, in which 0 corresponds to the motor stopped, and 100 to maximum speed.

- Similarly, to make the motor go backwards:

```
motor1.backwards(speed)
```

- It is also handy to have a command to make the motor stop, as follows:

```
motor1.stop()
```

Importing the dcmotor library

Create a new file called *dcmotor.py* and copy the following code into that file.

```
#This file includes a class to control DC motors

class DCMotor:
  #the min_duty and max_duty are defined for 15000Hz frequency
  #you can pass as arguments
  def __init__(self, pin1, pin2, enable_pin, min_duty=750,
max_duty=1023):
    self.pin1 = pin1
    self.pin2 = pin2
    self.enable_pin = enable_pin
    self.min_duty = min_duty
    self.max_duty = max_duty

  #speed value can be between 0 and 100
  def forward(self, speed):
    self.speed = speed
    self.enable_pin.duty(self.duty_cycle(self.speed))
    self.pin1.value(1)
    self.pin2.value(0)

  def backwards(self, speed):
    self.speed = speed
    self.enable_pin.duty(self.duty_cycle(self.speed))
    self.pin1.value(0)
    self.pin2.value(1)

  def stop(self):
    self.enable_pin.duty(0)
    self.pin1.value(0)
    self.pin2.value(0)
```

```
  def duty_cycle(self, speed):
    if self.speed <= 0 or self.speed > 100:
      duty_cycle = 0
    else:
      duty_cycle = int (self.min_duty + (self.max_duty - self.min_duty)*((self.speed - 1)/(100-1)))
    return duty_cycle
```

SOURCE CODE

https://github.com/RuiSantosdotme/ESP-MicroPython/blob/master/code/Others/DCMotor/dcmotor.py

Upload that file to your ESP32 or ESP8266 board with the *dcmotor.py* name.

How the *dcmotor* Module Works

The *dcmotor* module contains a single class called `DCMotor`.

```
class DCMotor:
```

We use the constructor method (`__init__`) to initiate the data as soon as an object of the `DCMotor` class is instantiated.

```
def __init__(self, pin1, pin2, enable_pin, min_duty=750, max_duty=1023):
  self.pin1 = pin1
  self.pin2 = pin2
  self.enable_pin = enable_pin
  self.min_duty = min_duty
  self.max_duty = max_duty
```

A `DCMotor` object accepts as parameters:

- **pin1**: GPIO (output) that connects to L298N input 1 pin.
- **pin2**: GPIO (output) that connects to L298N input 2 pin.
- **enable_pin**: GPIO (PWM pin) that connects to L298N enable 1 pin.
- **min_duty**: minimum duty cycle to make the motor move. This parameter is optional, and it's set to 750 by default. You may need to change this parameter depending on frequency required to control your DC motor.
- **max_duty**: maximum duty cycle to make the motor move. This parameter is set to 1023 by default.

Then, create several methods to control the DC motor: `forward()`, `backwards()`, and `stop()`.

The `forward()` and `backwards()` methods make the motor spin forward and backwards, respectively. The `stop()` method stops the motor.

The `forward()` and `backwards()` methods accepts the `speed` as a parameter. The `speed` should be an integer number between 0 and 100.

Let's take a closer look at the `forward()` method:

```
def forward(self, speed):
  self.speed = speed
  self.enable_pin.duty(self.duty_cycle(self.speed))
  self.pin1.value(1)
  self.pin2.value(0)
```

To make the motor move forward, set `pin1` to 1 and `pin2` to 0. The enable pin's duty cycle is set to the corresponding `speed`. The speed is an integer number between 0 and 100, but the duty cycle should be a number between the `max_duty` and `min_duty`. So, we have another method at the end of the code, called `duty_cycle()` that calculates the corresponding duty cycle value based on the speed.

```
def duty_cycle(self, speed):
  if self.speed <= 0 or self.speed > 100:
    duty_cycle = 0
  else:
    duty_cycle = int (self.min_duty + (self.max_duty -
self.min_duty)*((self.speed - 1)/(100-1)))
  return duty_cycle
```

The `backwards()` method works in a similar way, but `pin1` is set to 0, and `pin2` is set to 1.

```
def backwards(self, speed):
  self.speed = speed
  self.enable_pin.duty(self.duty_cycle(self.speed))
  self.pin1.value(0)
  self.pin2.value(1)
```

The `stop()` method sets the duty cycle , `pin1` and `pin2` values to 0.

```
def stop(self):
  self.enable_pin.duty(0)
  self.pin1.value(0)
  self.pin2.value(0)
```

Controlling the DC Motor

Now that we understand how the DC motor module works, we can create a new script to control the DC motor using the library functionalities.

The following code demonstrates how to use the library's functionalities to control the DC motor.

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

frequency = 15000

pin1 = Pin(12, Pin.OUT)
pin2 = Pin(14, Pin.OUT)
enable = PWM(Pin(13), frequency)

dc_motor = DCMotor(pin1, pin2, enable)

#Set min duty cycle (350) and max duty cycle (1023)
#dc_motor = DCMotor(pin1, pin2, enable, 350, 1023)

dc_motor.forward(50)
sleep(5)
dc_motor.stop()
sleep(5)
dc_motor.backwards(100)
sleep(5)
dc_motor.forward(5)
sleep(5)
dc_motor.stop()
```

SOURCE CODE
https://github.com/RuiSantosdotme/ESP-MicroPython/blob/master/code/Others/DCMotor/main_esp32.py

Importing libraries

To use the library's functionalities, you need to import it in your code. We import the `DCMotor` class from the `dcmotor` library we've created previously.

```
from dcmotor import DCMotor
```

You also need to import the `Pin` and `PWM` classes to control the GPIOs and the `sleep()` method.

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

Setting the frequency

Set the PWM signal frequency to 15000 Hz. You can choose other frequency values. Note that for lower frequency values, the motor may not spin and make a weird beep sound.

```
frequency = 15000
```

Initializing GPIOs

We create three variables that refer to the motor driver's pin1, pin2, and enable pins. These are connected to GPIO 12, 14, and 13, respectively. `pin1` and `pin2` are output pins, and the `enable` is a PWM pin.

```
pin1 = Pin(12, Pin.OUT)
pin2 = Pin(14, Pin.OUT)
enable = PWM(Pin(13), frequency)
```

Then, initialize a DCMotor object with the pin1, pin2, and enable defined earlier:

```
dc_motor = DCMotor(pin1, pin2, enable)
```

Note: you may need to pass the `min_duty` and `max_duty` parameters depending on the motor and board you're using. For example:

```
dc_motor = DCMotor(pin1, pin2, enable, 350, 1023)
```

Now you have a `DCMotor` object called `dc_motor`. You can use the methods to control the motor. Make the motor move forward at 50% speed:

```
dc_motor.forward(50)
```

Stop the motor:

```
dc_motor.stop()
```

Move the motor backwards at maximum speed:

```
dc_motor.backwards(100)
```

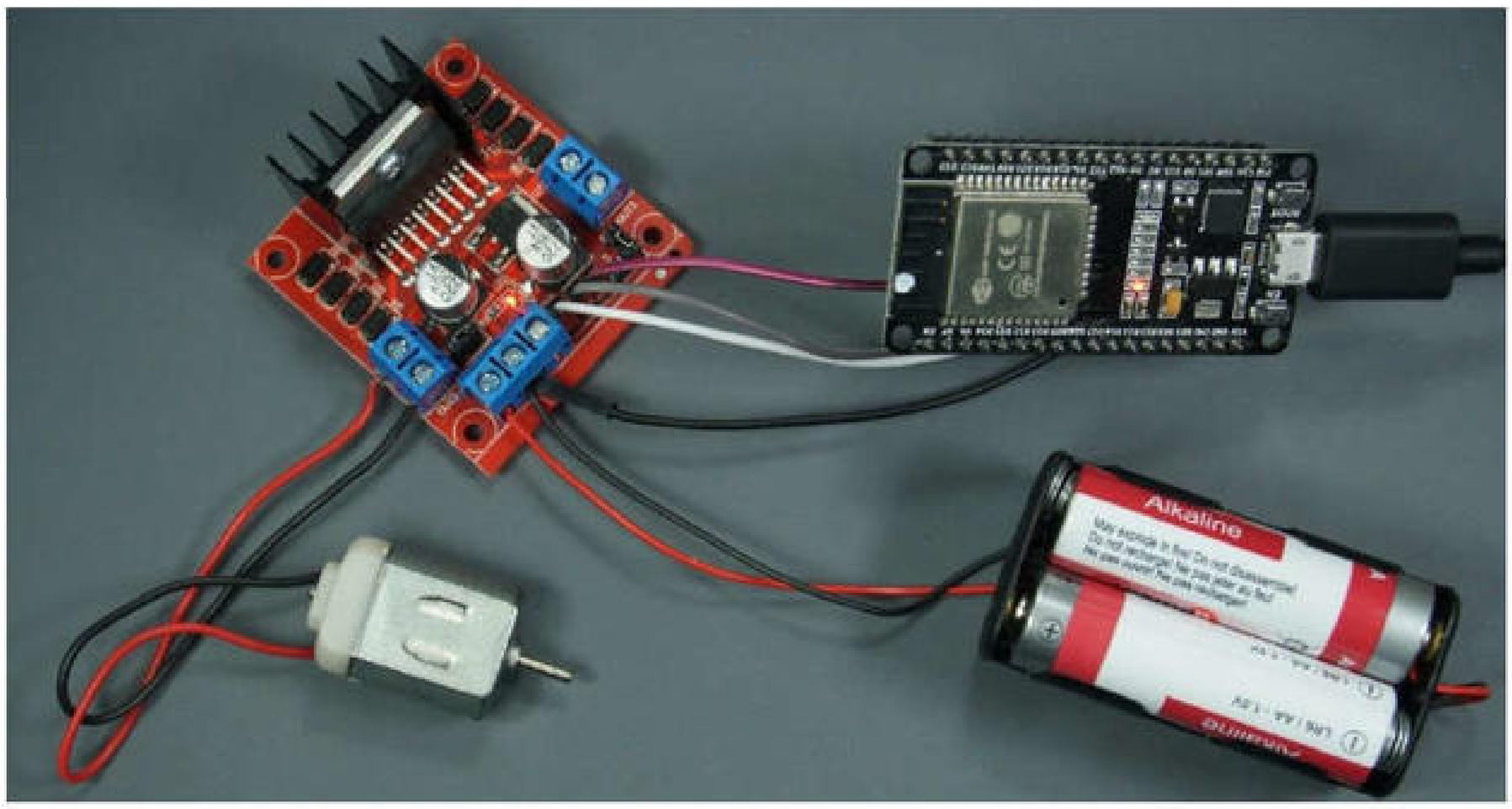
Move the motor forward at 5% speed:

```
dc_motor.forward(5)
```

You can experiment with the methods and pass different speeds as an argument to see how your DC motor behaves.

Wrapping Up

In this section you've learned how to control a DC motor using the L298N motor driver with the ESP32 or ESP8266. We've created a Python module with methods to easily control the motor: the *dcmotor.py* library.



Now, you can use this library in your projects to create `DCMotor` objects and use the library methods to control the motor: `forward()`, `backwards()` and `stop()`.

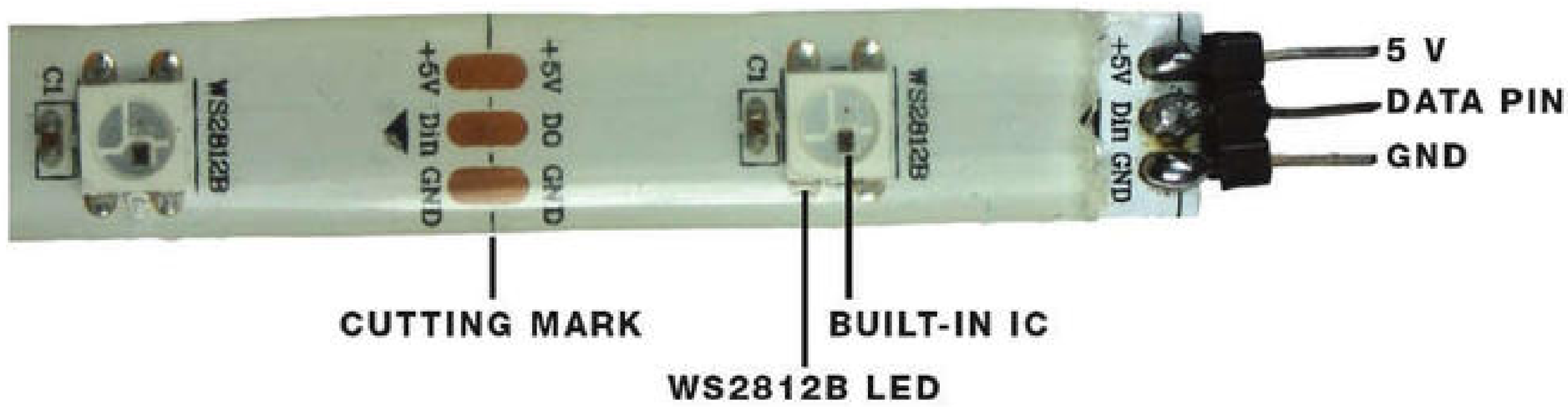
Control an Addressable RGB LED Strip

This Unit shows how to control an addressable RGB LED strip to make awesome lighting effects. We'll be using the WS2812B RGB LED strip, but other similar addressable RGB LED strips should also work.

WS2812B Addressable RGB LED Strip

When you first get the WS2812B strip, it comes in a reel, as shown in the following figure. You can cut off as long a section as you need.

The strip shown in the previous figure is 5 meters long with 300 addressable WS2812B RGB LEDs wired in series. We'll cut a section of 14 LEDs to use in this project. There are cutting marks, shown along the entire length of the strip.



The color and brightness of each LED can be controlled individually, allowing you to produce amazing effects easily. Each LED has an integrated circuit (IC) built right in, which means you can control the whole strip using just one GPIO pin, connected to the middle pin— the Data pin—at the end of the strip.

Prepare the LED strip for this project as follows:

1. Cut a strip of 14 LEDs along the cutting marks shown on the strip.
2. Solder header pins to the 5V, Data, and GND pins as shown in figure above.

Now you need to figure out your power supply. The LED strip requires a 5V power source. You can determine the amps you need from the amount of power each LED requires. An individual LED draws up to 60 mA at full brightness (which produces white light), but since you'll rarely need all LEDs at their maximum value for any length of time, you can safely estimate 20 mA per LED. So, if your strip is 14 LEDs long, you'll need a 5V power source with approximately 20 × 14 = 280 mA.

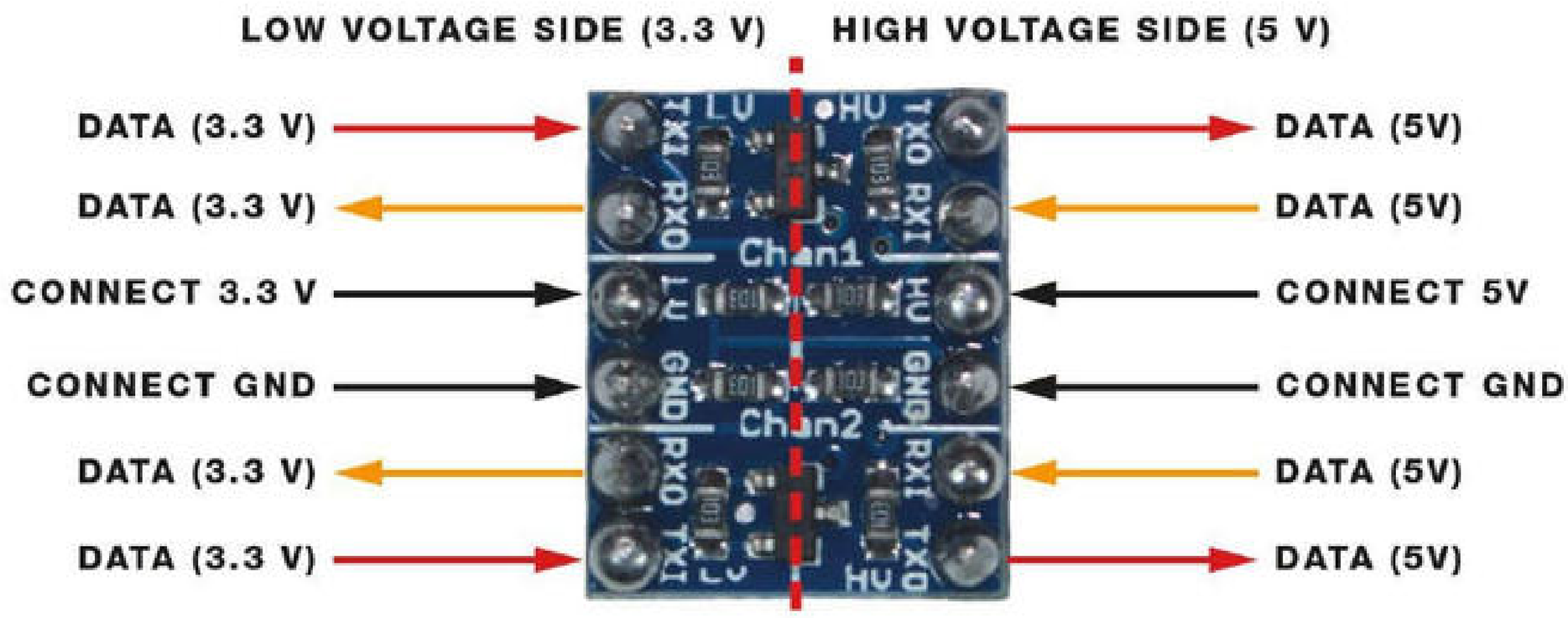
Note: powering your boards through the ESP32 or ESP8266 VIN pin works fine if you're controlling a few LEDs, like in this example. If you use more LEDs, you may need to use an external power supply because your board may not be able to supply enough current.

Introducing the Logic Level Converter

The strip Data pin needs a 5V signal, but the ESP32 and ESP8266 GPIOs operate at 3.3V. To get the 5V signal you need, you can use a component called a *logic level converter* (also called level-shifter).

Note: some ESP32/ESP8266 models can control the strip without the logic level converter. The ESP32 and ESP8266 boards we're using in this course worked fine without a logic level converter (DOIT ESP32 DEVKIT V1 Board and ESP8266 ESP-12E NodeMCU Kit). However, if your strip shows a weird behaviour when testing our examples, you'll probably need to add one to your circuit. We've recently found out that ESP32 boards from Adafruit needs a logic level converter. We provide schematic diagrams with and without logic level converter.

A logic level converter allows you to convert 3.3V signals to 5V signals. There are many types of logic level converter, but in this project we'll use the two-channel logic level converter bidirectional module shown in the following figure.



The bidirectionality of this module allows you to convert data in both ways—from 3.3V to 5V and from 5V to 3.3V. You won't need to convert 5V to 3.3V in this project but having this coventer is more flexible model in your toolkit (versus a unidirectional model) can come in handy for future projects. This logic level converter also has two channels, channel 1 and channel 2. In this project you'll use only one of the channels to control the LED strip's Data pin.

The module has a low-voltage side (left side), to which you attach everything that's at 3.3V, and a high-voltage side (right side), where you attach everything at 5V. For this project, you need to use one of the pins highlighted with a red arrow, as you want to send 3.3V data and convert it to 5V.

Schematic (with logic level converter)

This section shows how to wire the WS2812B LED strip to your ESP32 or ESP8266 boards using a logic level converter module. If you don't want to use a logic level converter module, follow the next section instead.

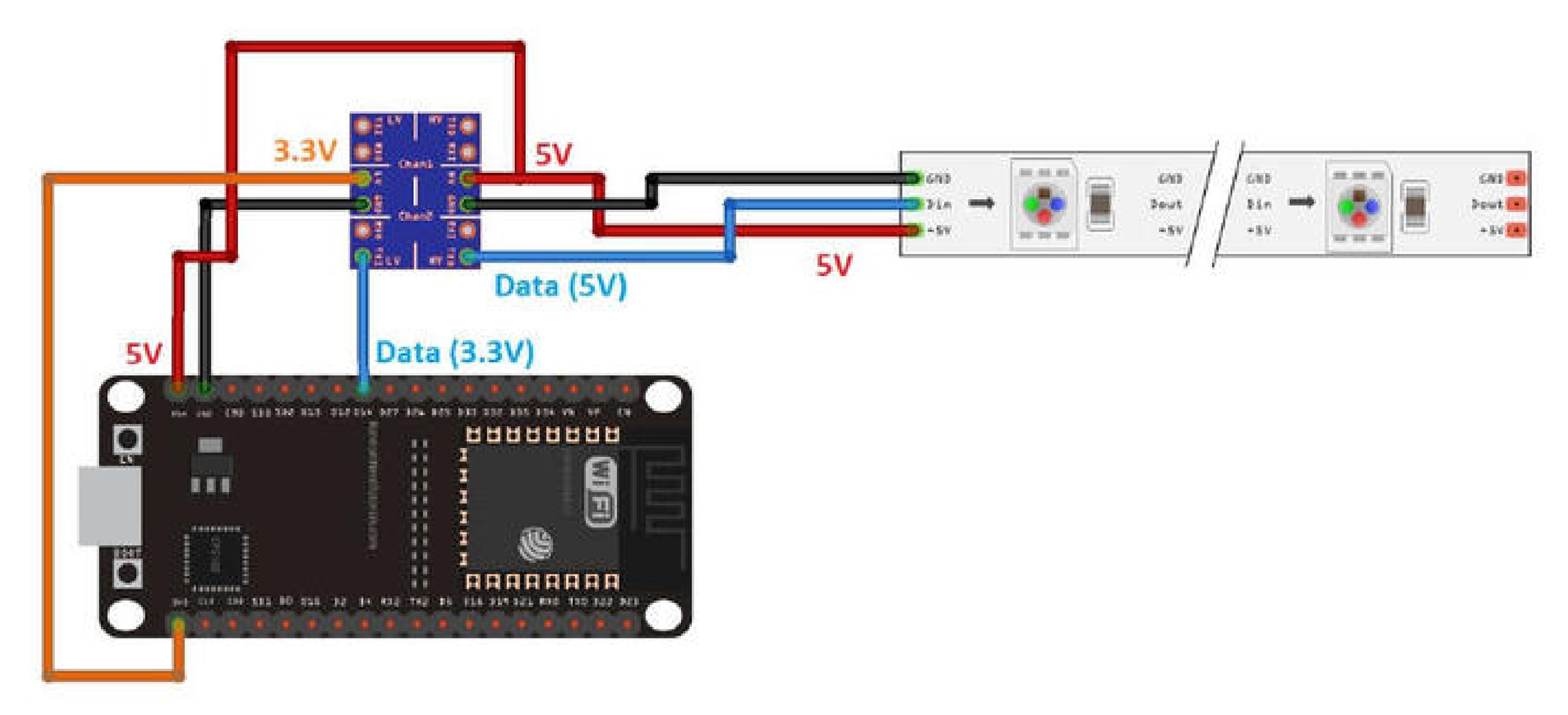
Parts Required

Here are the parts required for this project:

- ESP32 or ESP8266
- WS2812B addressable RGB LED strip
- Logic level converter module
- Jumper wires

ESP32

Follow the next schematic diagram if you're using an ESP32 board.



This schematic uses the ESP32 DEVKIT DOIT board version with 36 GPIOs. Before assembling the circuit double-check the pinout for the board you're using.

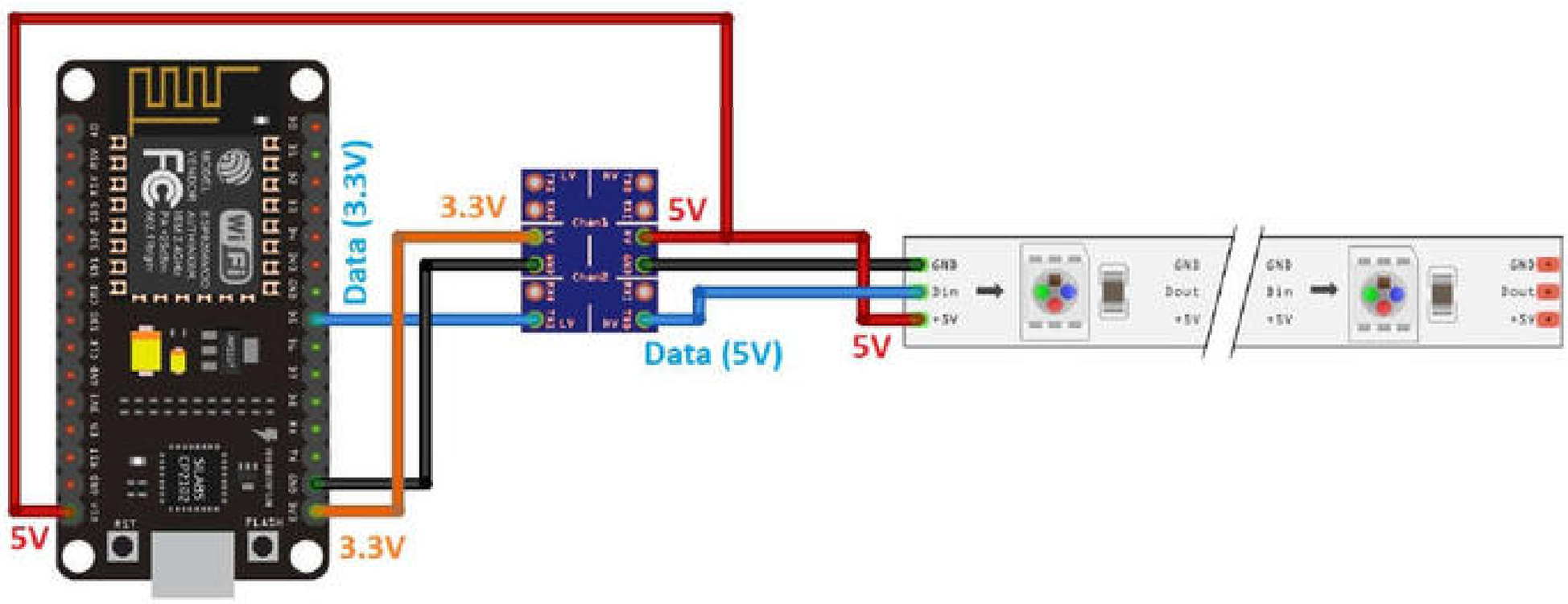
To use the logic level converter, connect GND on both sides, 3.3V on the low-voltage side, and 5V on the high-voltage side. Then, connect data from the ESP32 on one of the TX1 pins—you can use either channel 1 or channel 2—and get the 5V data signal on the corresponding TX0 pin.

For better understanding the connections, you can take a look at the following table:

Low-voltage side	
Logic Level Converter	**ESP32**
TX1 (channel 2)	GPIO 14
LV	3.3V
GND	GND
High-voltage side	
Logic Level Converter	**Connect to**
TX0 (channel 2)	RGB LED Strip's data pin
HV	5V
GND	GND

ESP8266

Follow the next schematic diagram if you're using an ESP8266 board.



To use the logic level converter, connect GND on both sides, 3.3V on the low-voltage side, and 5V on the high-voltage side. Then, connect data from the

ESP8266 on one of the TX1 pins—you can use either channel 1 or channel 2—and get the 5V data signal on the corresponding TX0 pin.

For better understanding the connections, you can take a look at the following table:

Low-voltage side	
Logic Level Converter	**ESP8266**
TX1 (channel 2)	GPIO 14 (D5)
LV	3.3V
GND	GND
High-voltage side	
Logic Level Converter	**Connect to**
TX0 (channel 2)	RGB LED Strip's data pin
HV	5V
GND	GND

Schematic (without logic level converter)

Wiring the RGB LED strip to the ESP32 or ESP8266 is very simple. You need to apply 5V to the VCC pin, GND to GND and connect a GPIO to the Din pin. We'll connect the data pin to GPIO14.

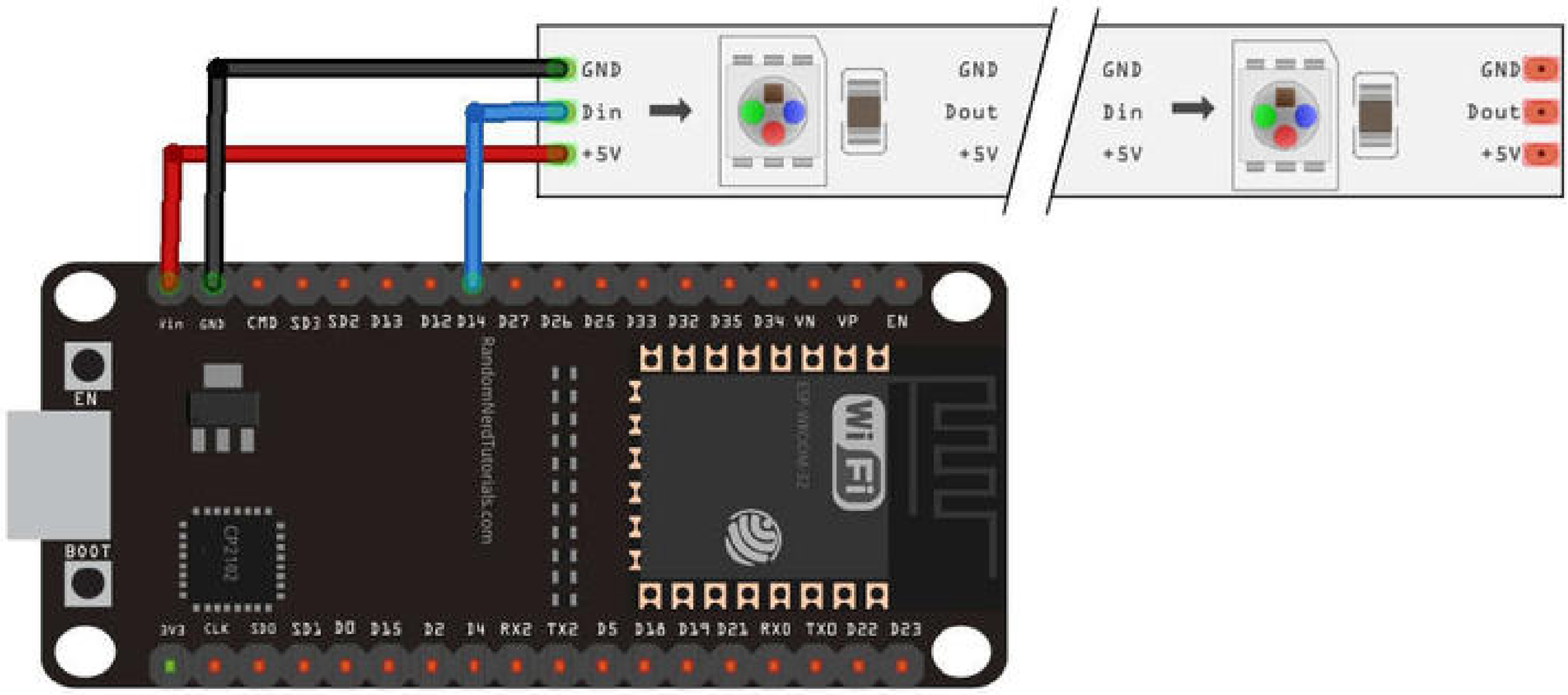
Parts Required

Here are the parts required for this project:

- ESP32 or ESP8266
- WS2812B addressable RGB LED strip
- Jumper wires

ESP32

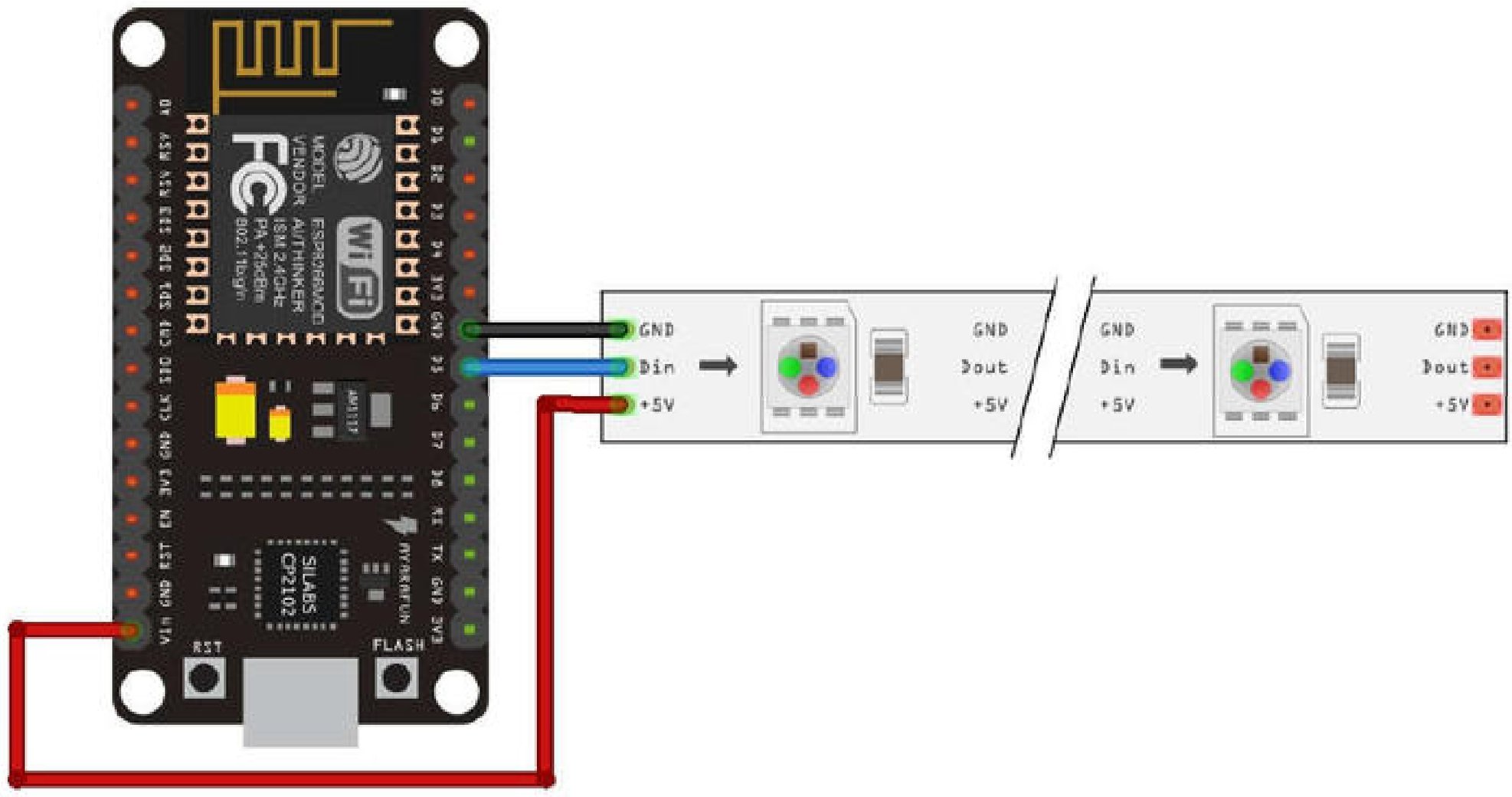
Follow the next schematic diagram if you're using an ESP32 board.



This schematic uses the ESP32 DEVKIT DOIT board version with 36 GPIOs. Before assembling the circuit double-check the pinout for the board you're using.

ESP8266

Follow the next schematic diagram if you're using an ESP8266 board.



Script

There's a built-in MicroPython module called *neopixel* that makes it very easy to control the RGB LED strip. The following script shows how to control each pixel of the LED strip and provides functions with awesome lighting effects.

Copy the following code to your ESP32/ESP8266 *main.py* file.

```
import machine, neopixel
import time

# number of pixels
n = 14
# strip control gpio
p = 14

np = neopixel.NeoPixel(machine.Pin(p), n)

# set single pixel (1st pixel = index [0]) to red color
np[0] = (255, 0, 0)
np.write()
time.sleep(1)
```

```
# set strip color
def set_color(r, g, b):
  for i in range(n):
    np[i] = (r, g, b)
    np.write()

set_color(0, 0, 255)
time.sleep(1)

# fade in/out
def fade_in_out(color, wait):
  for i in range(0, 4 * 256, 8):
    for j in range(n):
      if (i // 256) % 2 == 0:
        val = i & 0xff
      else:
        val = 255 - (i & 0xff)
        if color == 'red':
          np[j] = (val, 0, 0)
        elif color == 'green':
          np[j] = (0, val, 0)
```

```
        elif color == 'blue':
          np[j] = (0, 0, val)
        elif color == 'purple':
          np[j] = (val, 0, val)
        elif color == 'yellow':
          np[j] = (val, val, 0)
        elif color == 'teal':
          np[j] = (0, val, val)
        elif color == 'white':
          np[j] = (val, val, val)
      np.write()
    time.sleep_ms(wait)

#fade_in_out('red', 0)
fade_in_out('green', 10)
#fade_in_out('blue', 25)
#fade_in_out('purple', 10)
fade_in_out('yellow', 10)
fade_in_out('teal', 10)
#fade_in_out('white', 10)
time.sleep(1)

# bounce
def bounce(r, g, b, wait):
  for i in range(4 * n):
    for j in range(n):
      np[j] = (r, g, b)
    if (i // n) % 2 == 0:
      np[i % n] = (0, 0, 0)
    else:
      np[n - 1 - (i % n)] = (0, 0, 0)
    np.write()
    time.sleep_ms(wait)

bounce(255, 0, 125, 50)
time.sleep(1)

# cycle
def cycle(r, g, b, wait):
  for i in range(4 * n):
    for j in range(n):
      np[j] = (0, 0, 0)
    np[i % n] = (r, g, b)
    np.write()
    time.sleep_ms(wait)
```

```
cycle(0, 255, 0, 50)
time.sleep(1)

# function to go through all colors
def wheel(pos):
  # Input a value 0 to 255 to get a color value.
  # The colours are a transition r - g - b - back to r.
  if pos < 0 or pos > 255:
    return (0, 0, 0)
  if pos < 85:
    return (255 - pos * 3, pos * 3, 0)
  if pos < 170:
    pos -= 85
    return (0, 255 - pos * 3, pos * 3)
  pos -= 170
  return (pos * 3, 0, 255 - pos * 3)

# rainbow
def rainbow_cycle(wait):
  for j in range(255):
    for i in range(n):
      rc_index = (i * 256 // n) + j
      np[i] = wheel(rc_index & 255)
    np.write()    time.sleep_ms(wait)
```

```
rainbow_cycle(10)
rainbow_cycle(5)
time.sleep(1)

# turn off all pixels
def clear():
  for i in range(n):
    np[i] = (0, 0, 0)
    np.write()

clear()
```

SOURCE CODE
https://github.com/RuiSantosdotme/ESP-MicroPython/blob/master/code/Others/WS2812B/main.py

Importing libraries

You start by importing the necessary libraries: the `neopixel` library to control the strip, the `machine` module to set up the GPIO and the `time` module to add delays.

```
import machine, neopixel
import time
```

Creating a NeoPixel object

You set the strip number of pixels in the `n` variable:

```
n = 14
```

Save the GPIO number that will control the strip on the `p` variable:

```
p = 14
```

Create a `NeoPixel` object called `np` on the GPIO you've defined earlier and with the number of LEDs you've also defined:

```
np = neopixel.NeoPixel(machine.Pin(p), n)
```

Controlling an individual pixel

Controlling an individual pixel is very easy. You can think of the strip as an array with 14 elements (number of pixels on our sample strip). Then, we just need to set a colour to a specific element. For example, to set the first pixel to red:

```
np[0] = (255, 0, 0)
```

Then, use the `write()` method for the changes to take effect.

```
np.write()
```

If you want to set pixel number 3 with blue, you use the following code instead:

```
np[2] = (0, 0, 255)
np.write()
```

Set all pixels to the same color

After knowing how to control individual pixels, it is easy to make some lighting effects. We've created some functions (adapted from the library examples) with lighting effects, you can copy to use in other projects.

For example, the following snippet creates a function called `set_color` that accepts as arguments: `r`, `g` and `b` parameters to set the color. This function sets all the pixels on the strip to the same color.

```
# set strip color
def set_color(r, g, b):
  for i in range(n):
    np[i] = (r, g, b)
    np.write()
```

You can call the function as follows:

```
set_color(0, 0, 255)
```

It sets all pixels to blue color.

Fade in/ fade out effect

Then, there is the `fade_in_out` function. This function accepts as parameters: color and waiting time. The color can be one of the following options:

- 'red'
- 'green'
- 'blue'
- 'purple'
- 'yellow'
- 'teal'
- 'white'

The waiting time defines how fast the color fades in and out.

```
def fade_in_out(color, wait):
  for i in range(0, 4 * 256, 8):
    for j in range(n):
      if (i // 256) % 2 == 0:
        val = i & 0xff
      else:
        val = 255 - (i & 0xff)
        if color == 'red':
          np[j] = (val, 0, 0)
        elif color == 'green':
          np[j] = (0, val, 0)
```

```
        elif color == 'blue':
          np[j] = (0, 0, val)
        elif color == 'purple':
          np[j] = (val, 0, val)
        elif color == 'yellow':
          np[j] = (val, val, 0)
        elif color == 'teal':
          np[j] = (0, val, val)
        elif color == 'white':
          np[j] = (val, val, val)
      np.write()
    time.sleep_ms(wait)
```

Call that function using different parameters to test how it works. For example:

```
fade_in_out('red, 5)
fade_in_out('green', 10)
fade_in_out('blue', 25)
fade_in_out('purple', 30)
fade_in_out('yellow', 10)
fade_in_out('teal', 10)
fade_in_out('white', 10)
```

Bounce effect

The bounce function creates a bounce effect and accepts `r`, `g` and `b` parameters to set the color and the waiting time.

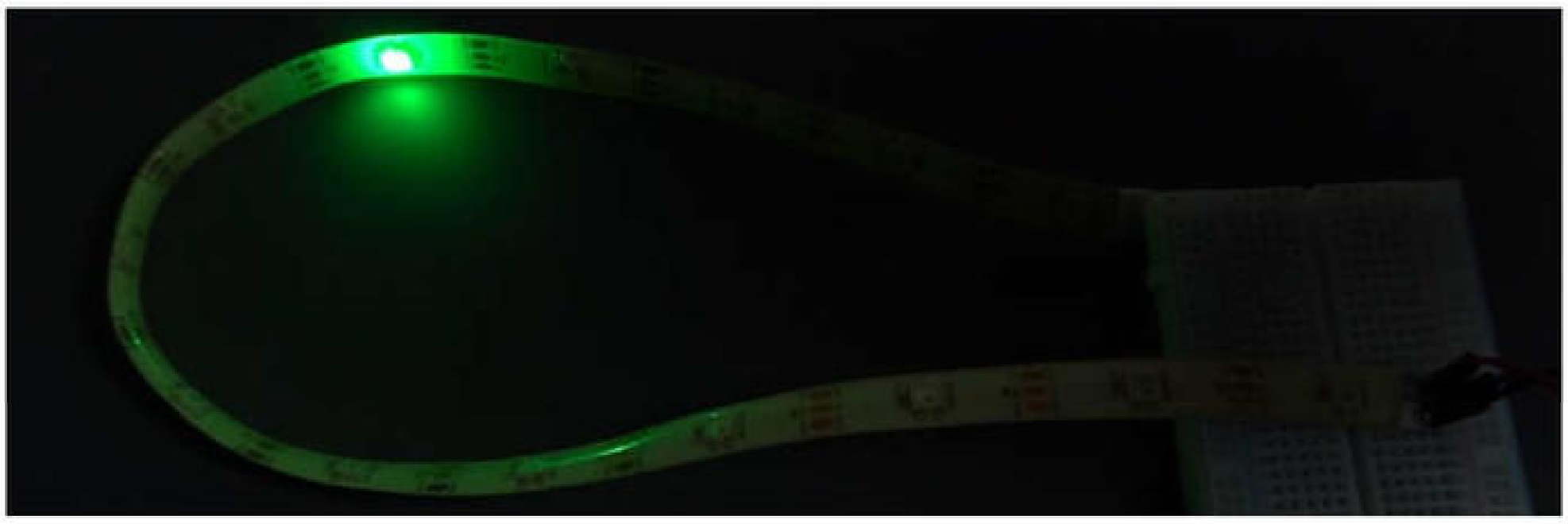
```
def bounce(r, g, b, wait):
  for i in range(4 * n):
    for j in range(n):
      np[j] = (r, g, b)
    if (i // n) % 2 == 0:
      np[i % n] = (0, 0, 0)
    else:
      np[n - 1 - (i % n)] = (0, 0, 0)
    np.write()
    time.sleep_ms(wait)
```

This effect shows an off pixel that runs through all the strip positions. Showing how this effect works on a static image is a bit difficult. We provide a link to a video later in this section that shows all the effects.

Cycle effect

The cycle effect works similarly to the bounce effect. There is a pixel on that runs through all the strip positions while the other pixels are off.

```
def cycle(r, g, b, wait):
  for i in range(4 * n):
    for j in range(n):
      np[j] = (0, 0, 0)
    np[i % n] = (r, g, b)
    np.write()
    time.sleep_ms(wait)
```



Moving rainbow effect

Finally, there are two functions to create a moving rainbow effect. The `wheel()` function generates the rainbow color spectrum by varying each color parameter between 0 and 255.

Each color is composed of red, green, and blue (RGB) parameters, and varying each parameter between 0 and 255 produces different colors, resulting in a rainbow effect.

```
def wheel(pos):
  # Input a value 0 to 255 to get a color value.
  # The colours are a transition r - g - b - back to r.
  if pos < 0 or pos > 255:
    return (0, 0, 0)
  if pos < 85:
    return (255 - pos * 3, pos * 3, 0)
  if pos < 170:
    pos -= 85
    return (0, 255 - pos * 3, pos * 3)
  pos -= 170
  return (pos * 3, 0, 255 - pos * 3)
```

Then, the `rainbowCycle()` function uses the results from the `wheel()` function to distribute the rainbow across the number of LEDs on your strip.

```
def rainbow_cycle(wait):
  for j in range(255):
    for i in range(n):
      rc_index = (i * 256 // n) + j
      np[i] = wheel(rc_index & 255)
    np.write()
    time.sleep_ms(wait)

rainbow_cycle(10)
rainbow_cycle(5)
time.sleep(1)
```

Clearing the strip

Finally, there is also a function to turn off all neopixels: the `clear()` function. This is the same as setting the LED strip to black color:

```
def clear():
  for i in range(n):
    np[i] = (0, 0, 0)
    np.write()
```

Demonstration

If you want to see a live demonstration of all the RGB LED strip color effects, click the following link to watch the video: https://youtu.be/vVOeDXugQRg

Wrapping Up

Controlling an addressable RGB LED strip is easy thanks to the `neopixel` library. The strip is an array, and each pixel is an element. Then, you just need to set a specific element to an RGB color.

We've shown you how to control individual pixels, as well as several functions to make lighting effects. You can apply these functions to your own projects.

MODULE 7

Thonny IDE

If you've had trouble using uPyCraft IDE, or if you simply want to experiment with Thonny IDE, follow this Module. We'll show you how to install it in your computer, how to flash firmware, and how to write and upload code to your boards. After completing this guide, you'll have your first LED blinking using MicroPython and Thonny IDE.

We've experimented with several IDEs to program the ESP32 and ESP8266 boards using MicroPython, and Thonny IDE seemed a good choice. Although there are some bugs, it is constantly being updated and improved. It allows you to program your ESP32 and ESP8266 boards with MicroPython, and it's compatible with Windows, Mac OS X, and Linux. It even comes installed by default with Raspbian OS for Raspberry Pi. Additionally, it's easy to install, so you shouldn't have problems with the installation process.

Before experimenting with Thonny IDE, you should flash your boards with MicroPython firmware. We provide an installation process using the *esptool.py* software. After that, there is a getting started guide for Thonny IDE.

Flashing MicroPython Firmware (esptool.py)

Unlike other boards, MicroPython isn't flashed onto the ESP32 or ESP8266 by default. That's the first thing you need to do to start programming your boards with MicroPython: flash/upload the firmware.

Tip: for an easier method to flash the ESP32/ESP8266 with MicroPython, you can flash MicroPython with uPyCraft IDE – see Module 1. However, we've recently received comments from our readers about having trouble installing/using uPyCraft IDE. For that reason, we're showing an alternative method using the *esptool.py* software.

Installing esptool.py in your computer

To work with *esptool.py*, you'll need either Python 2.7, Python 3.4 or a newer Python installation on your system. We recommend using Python 3.7.X, so go to Python's website and install it in your computer.

With Python 3 installed, open a Terminal window and install the latest stable esptool.py release with pip by typing the following:

```
pip install esptool
```

Note: with some Python installations, that command may not work, and you'll receive an error. If that's the case, try to install esptool.py with:

- `pip3 install esptool`
- `python -m pip install esptool`
- `pip2 install esptool`

After installing, you will have *esptool.py* installed into the default Python executables directory and you should be able to run it with the command `esptool.py`. In your Terminal window, run the following command:

```
esptool.py
```

If it is installed properly, it should display a similar message (regardless of your operating system):

```
Command Prompt
C:\Users\ruisantos>esptool.py
esptool.py v2.6
usage: esptool [-h] [--chip {auto,esp8266,esp32}] [--port PORT] [--baud BAUD]
               [--before {default_reset,no_reset,no_reset_no_sync}]
               [--after {hard_reset,soft_reset,no_reset}] [--no-stub]
               [--trace] [--override-vddsdio [{1.8V,1.9V,OFF}]]
               {load_ram,dump_mem,read_mem,write_mem,write_flash,run,image_info,make_image,elf2image,read_mac,chip_id,flash_id,rea
d_flash_status,write_flash_status,read_flash,verify_flash,erase_flash,erase_region,version}
               ...

esptool.py v2.6 - ESP8266 ROM Bootloader Utility

positional arguments:
  {load_ram,dump_mem,read_mem,write_mem,write_flash,run,image_info,make_image,elf2image,read_mac,chip_id,flash_id,read_flash_statu
s,write_flash_status,read_flash,verify_flash,erase_flash,erase_region,version}
                        Run esptool {command} -h for additional help
    load_ram            Download an image to RAM and execute
    dump_mem            Dump arbitrary memory to disk
    read_mem            Read arbitrary memory location
    write_mem           Read-modify-write to arbitrary memory location
    write_flash         Write a binary blob to flash
    run                 Run application code in flash
    image_info          Dump headers from an application image
    make_image          Create an application image from binary files
    elf2image           Create an application image from ELF file
    read_mac            Read MAC address from OTP ROM
    chip_id             Read Chip ID from OTP ROM
    flash_id            Read SPI flash manufacturer and device ID
    read_flash_status   Read SPI flash status register
    write_flash_status  Write SPI flash status register
    read_flash          Read SPI flash content
    verify_flash        Verify a binary blob against flash
    erase_flash         Perform Chip Erase on SPI flash
    erase_region        Erase a region of the flash
    version             Print esptool version

optional arguments:
  -h, --help            show this help message and exit
  --chip {auto,esp8266,esp32}, -c {auto,esp8266,esp32}
                        Target chip type
  --port PORT, -p PORT  Serial port device
  --baud BAUD, -b BAUD  Serial port baud rate used when flashing/reading
  --before {default_reset,no_reset,no_reset_no_sync}
                        What to do before connecting to the chip
  --after {hard_reset,soft_reset,no_reset}, -a {hard_reset,soft_reset,no_reset}
                        What to do after esptool.py is finished
  --no-stub             Disable launching the flasher stub, only talk to ROM
                        bootloader. Some features will not be available.
  --trace, -t           Enable trace-level output of esptool.py interactions.
  --override-vddsdio [{1.8V,1.9V,OFF}]
                        Override ESP32 VDDSDIO internal voltage regulator (use
                        with care)

C:\Users\ruisantos>
```

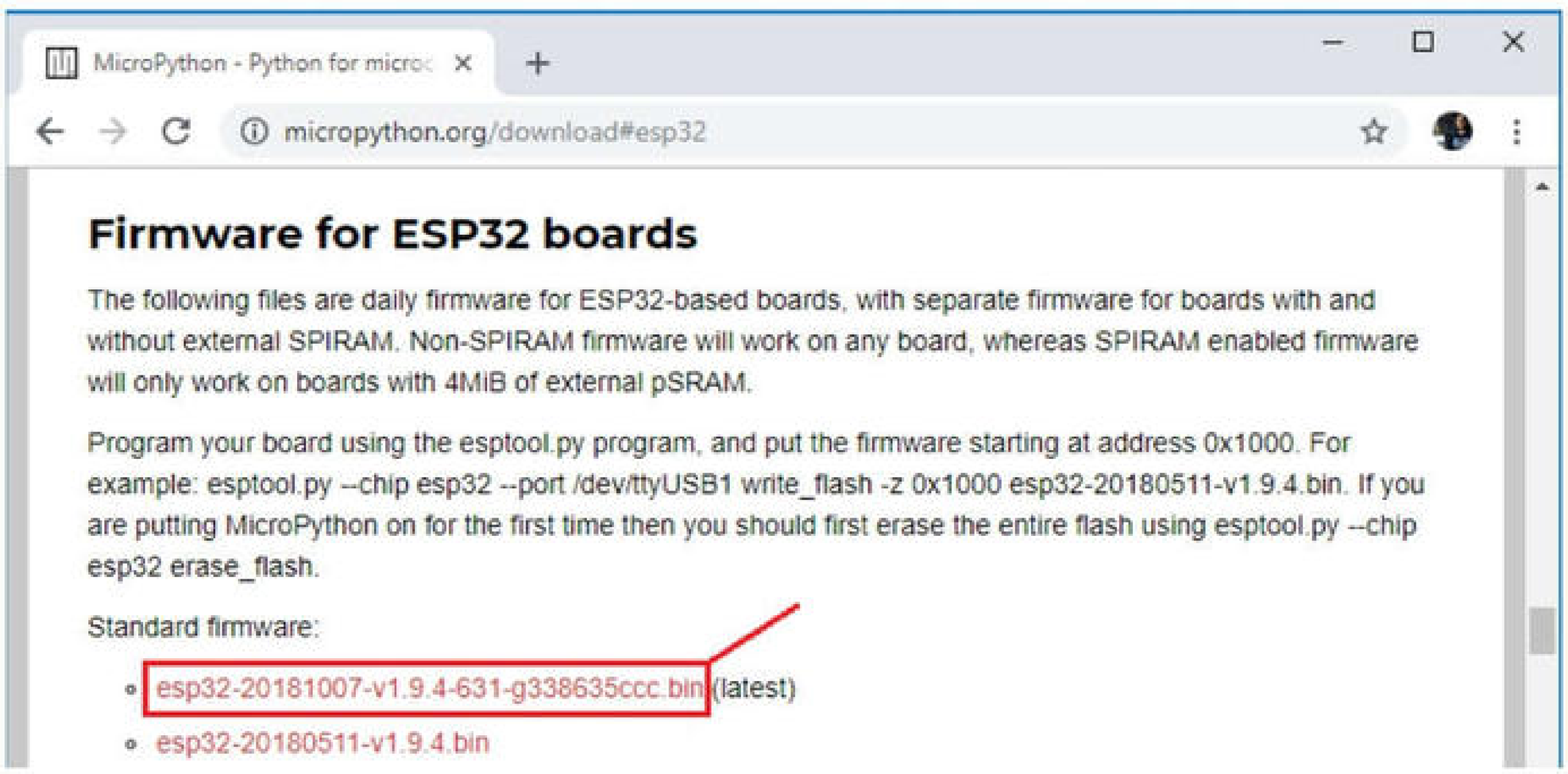
With *esptool.py* installed in your computer, you can easily flash your ESP32 or ESP8266 boards with the MicroPython firmware. Follow one of the next parts depending on your board:

- Part 1 – ESP32
- Part 2 – ESP8266

[Part 1 – ESP32] Downloading and Flashing the MicroPython Firmware on ESP32

To download the latest version of MicroPython firmware for the ESP32, go to the MicroPython Downloads page and scroll all the way down to the ESP32 section.

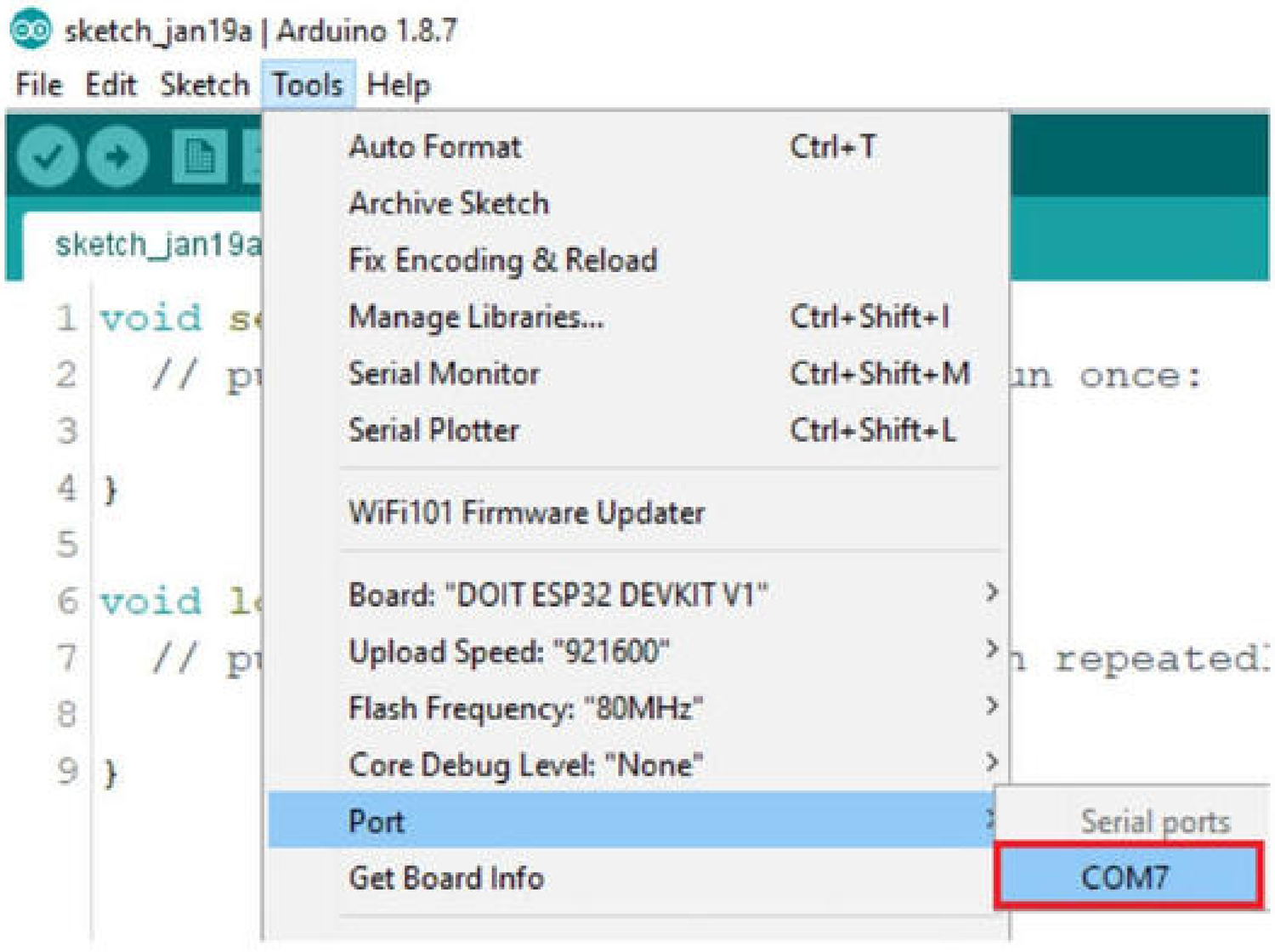
You should see a similar web page (see figure below) with the latest link to download the ESP32 .bin file – for example: esp32-20181007-v1.9.4-631-g338635ccc.bin.



Finding the Serial Port Name

It's a bit different to find the Serial port name in each operating system, so for simplicity reasons we recommend finding your ESP serial port name through the Arduino IDE. Follow these steps:

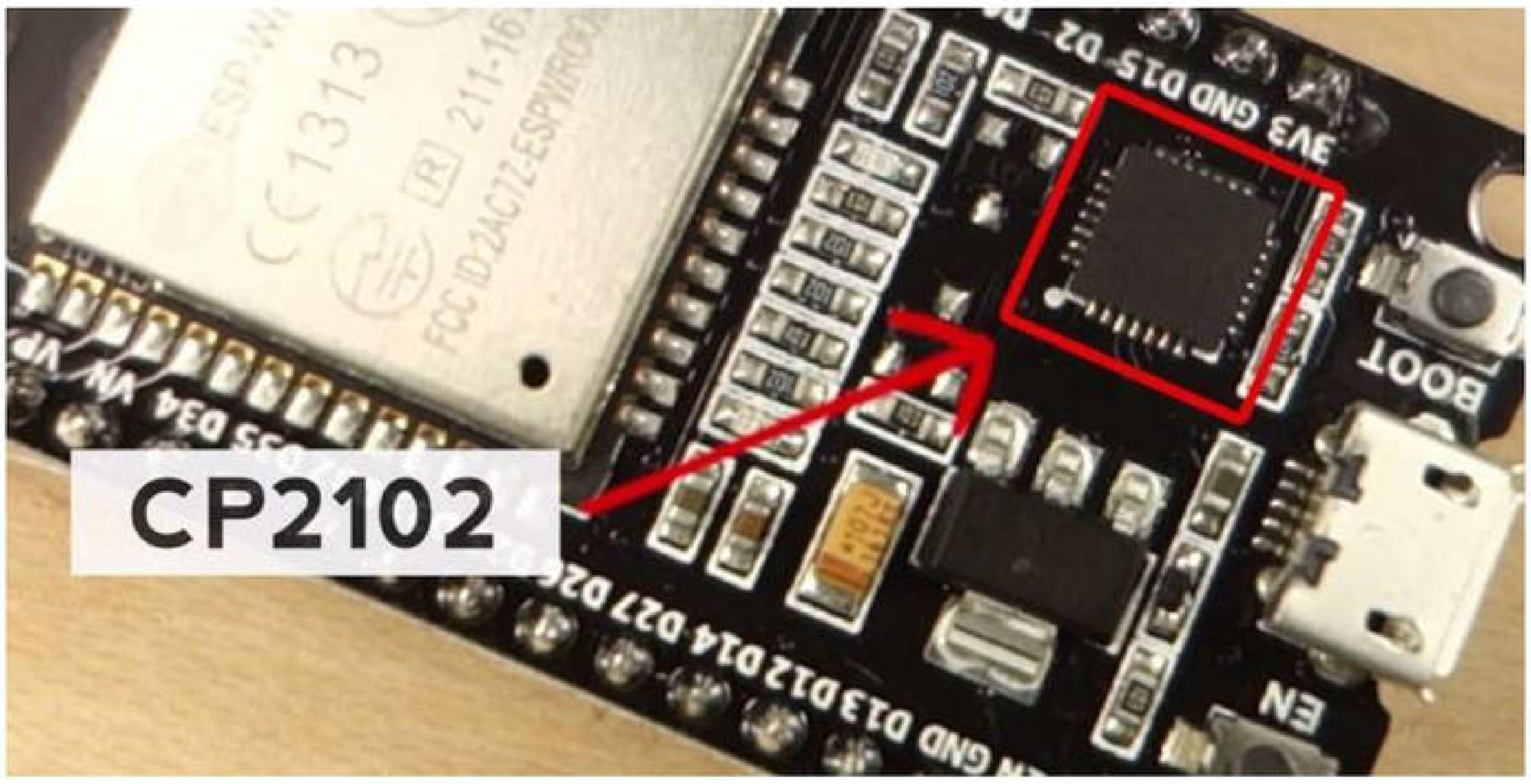
1. Connect your board to your computer
2. Open the Arduino IDE
3. Go to **Tools** ▸ **Port**
4. Save your ESP32 serial port name (in our case it's COM7)
5. Close your Arduino IDE software



Important: if you plug your ESP32 board to your computer, but you can't find the ESP32 Port available in your Arduino IDE, it might be one of these two problems: **1.** USB drivers missing or **2.** USB cable without data wires.

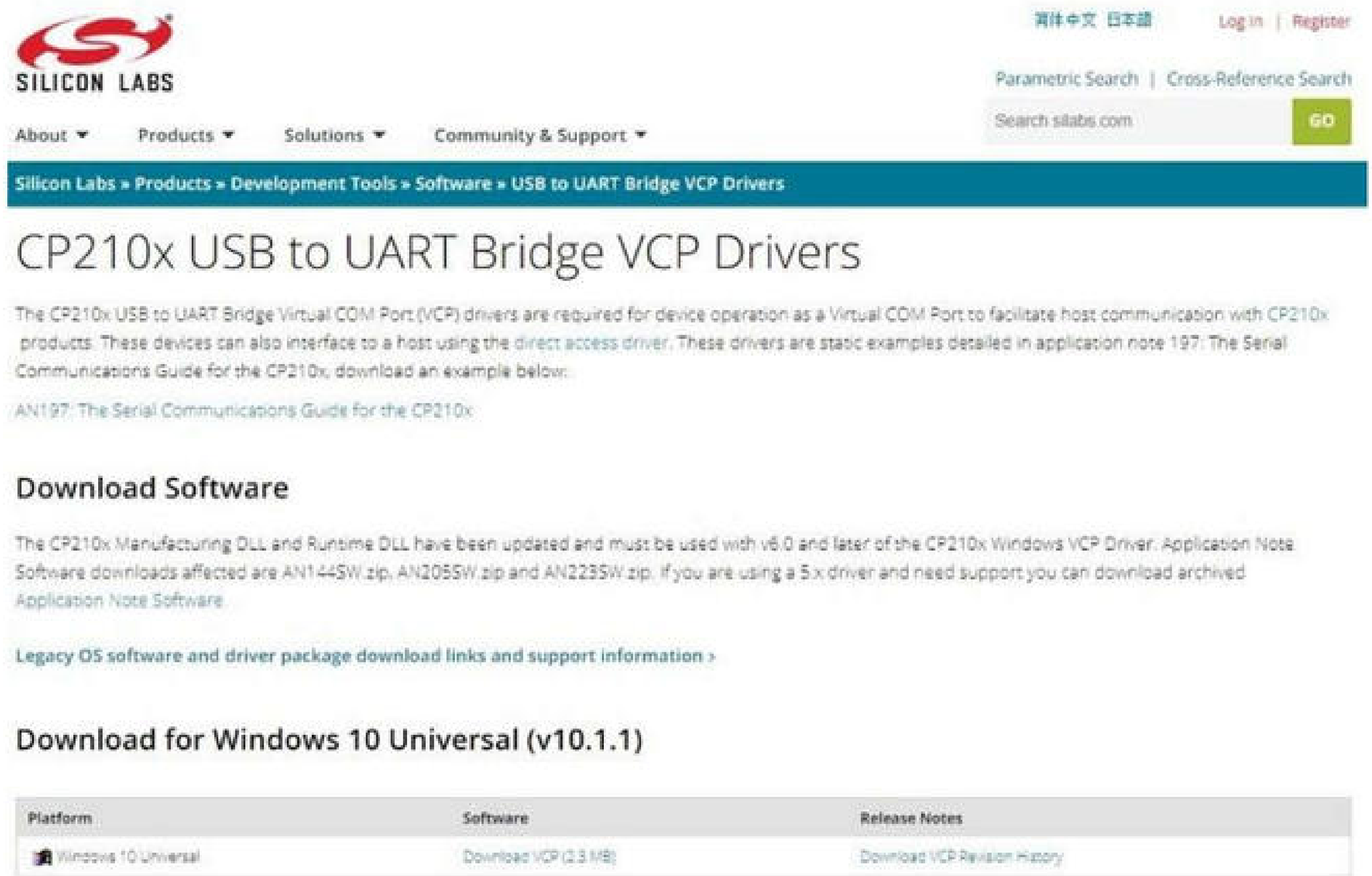
1. If you don't see your ESP's COM port available, this often means you don't have the USB drivers installed. Take a closer look at the chip next to the voltage regulator on board and check its name.

The ESP32 DEVKIT V1 DOIT board uses the CP2102 chip.



Go to Google and search for your specific chip to find the drivers and install them in your operating system.

You can download the CP2102 drivers on the Silicon Labs website.



After they are installed, restart the Arduino IDE and you should see the serial port in the **Tools ▸ Port** menu.

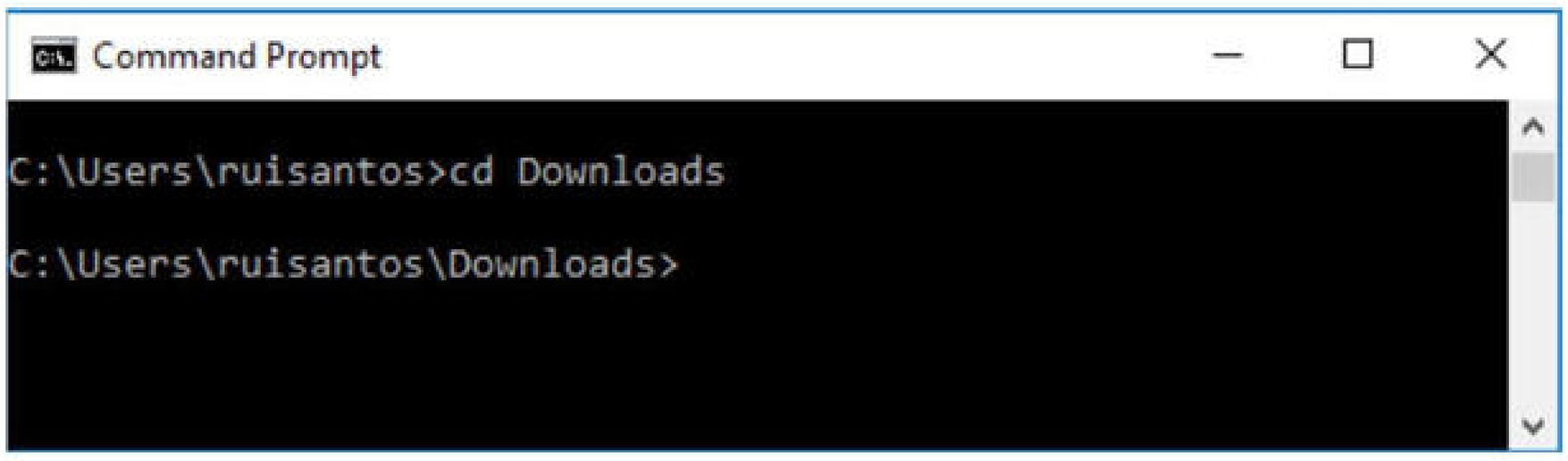
2. If you have the drivers installed, but you can't see your device, double-check that you're using a USB cable with data wires.

USB cables from powerbanks often don't have data wires (they are charge only). So, your computer will never establish a serial communication with your ESP32. Using a proper USB cable should solve your problem.

Finding your MicroPython .bin file

After downloading the ESP32 *.bin* file, it should be in your **Downloads** folder. So, with your Terminal window, you'll need to navigate to the **Downloads** folder using the cd command:

```
cd Downloads
```



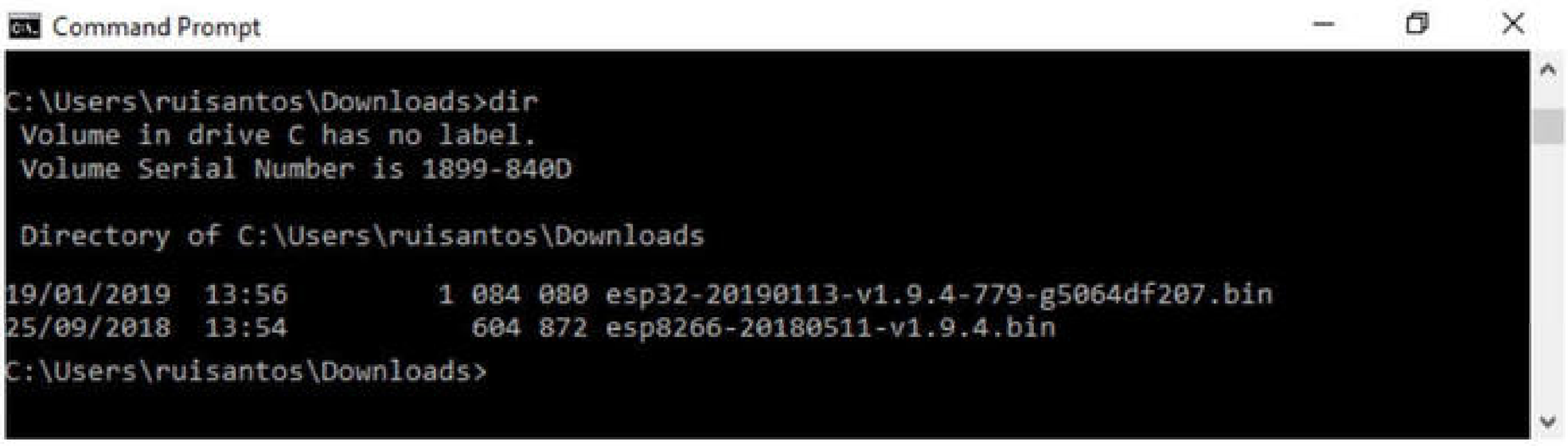
List all files in your **Downloads** folder to ensure that's where the *.bin* file is located.

In Windows, you use:

```
dir
```

On Mac OS X or Linux, run the next command:

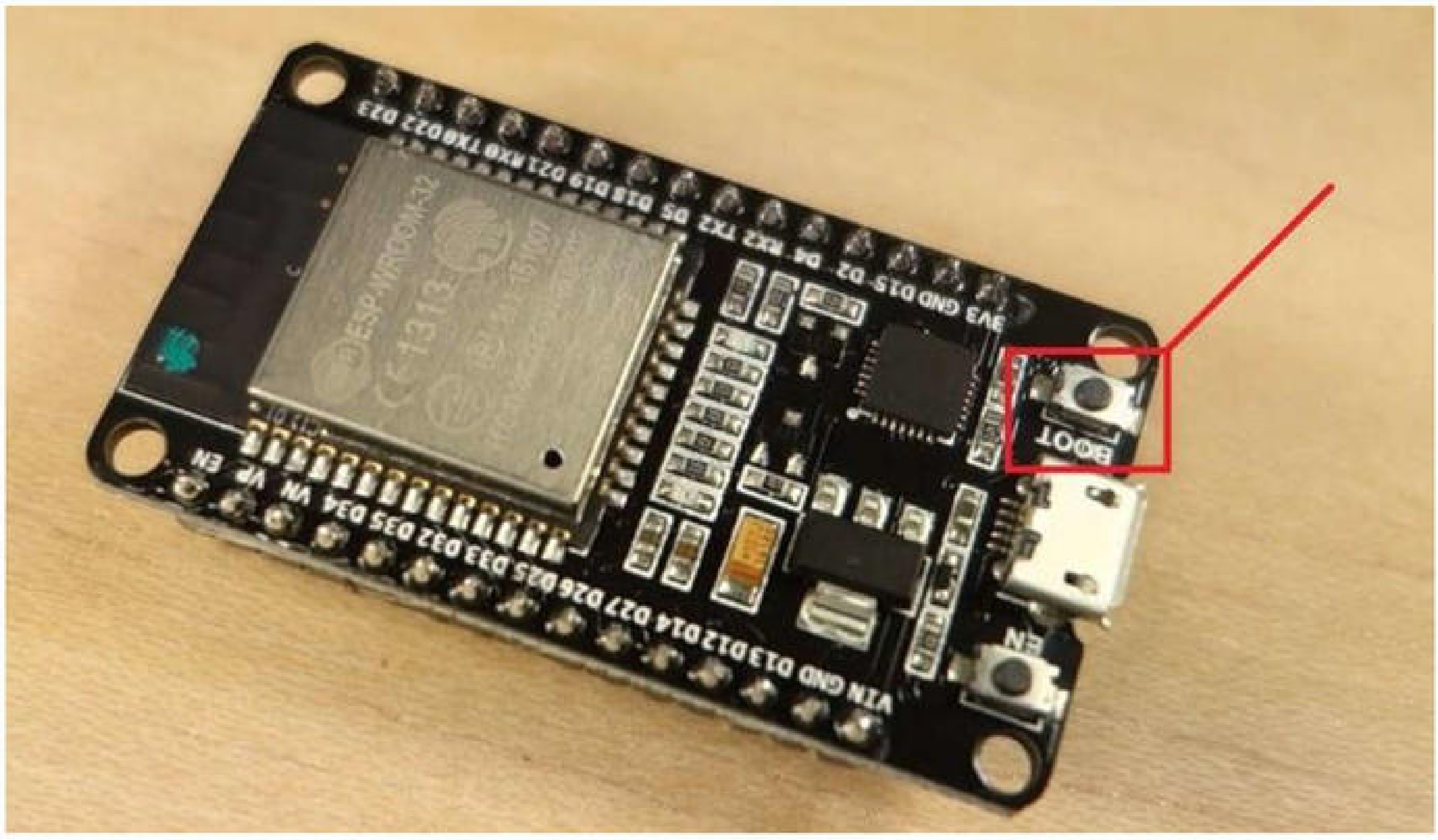
```
ls
```



As you can see in the preceding screenshot, the ESP32 *.bin* file is located in the **Downloads** folder: *esp32-20190113-v1.9.4-779-g5064df207.bin*.

Erasing ESP32 Flash Memory

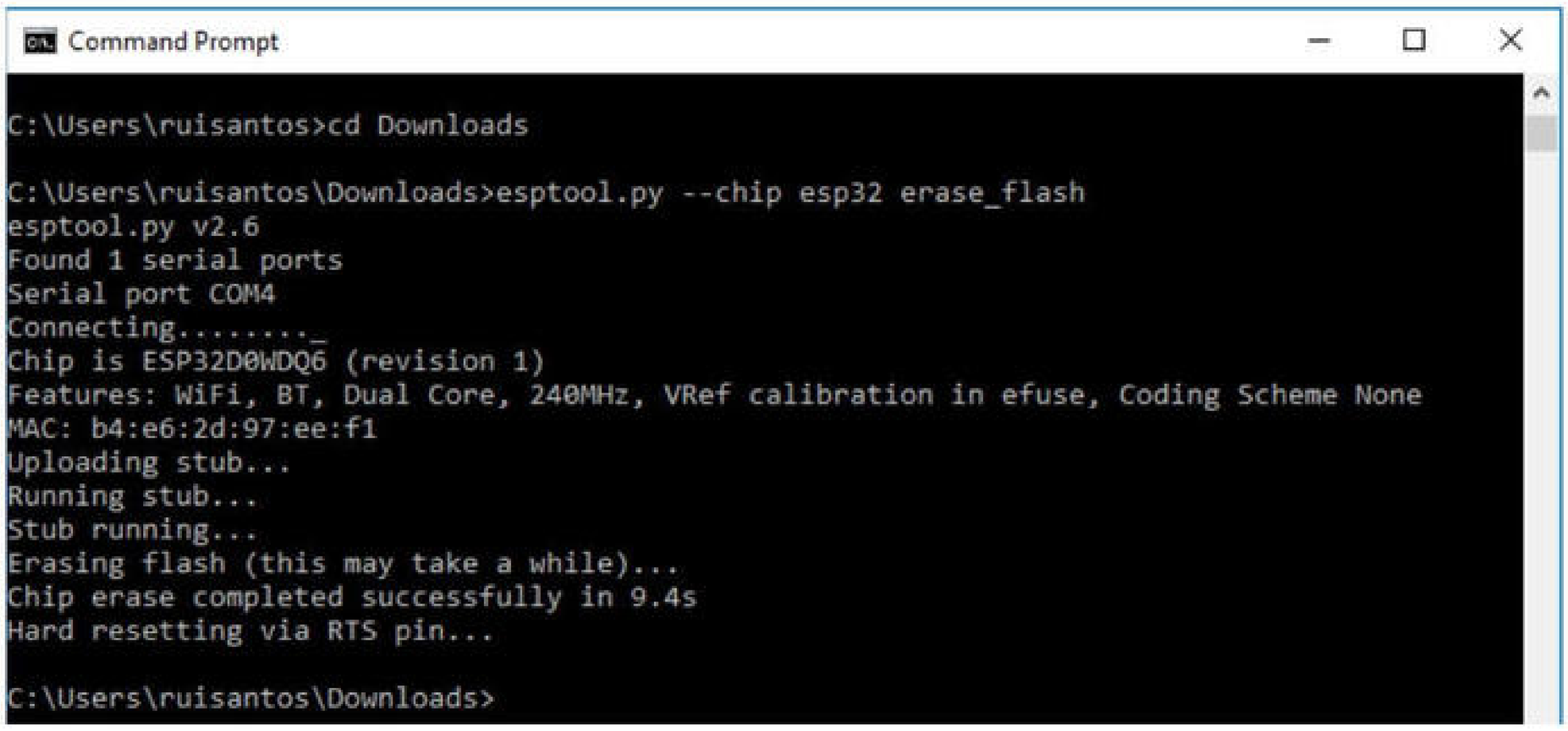
Before flashing the MicroPython firmware, you need to erase the ESP32 flash memory. So, with your ESP32 connected to your computer, hold-down the "BOOT/FLASH" button in your ESP32 board:



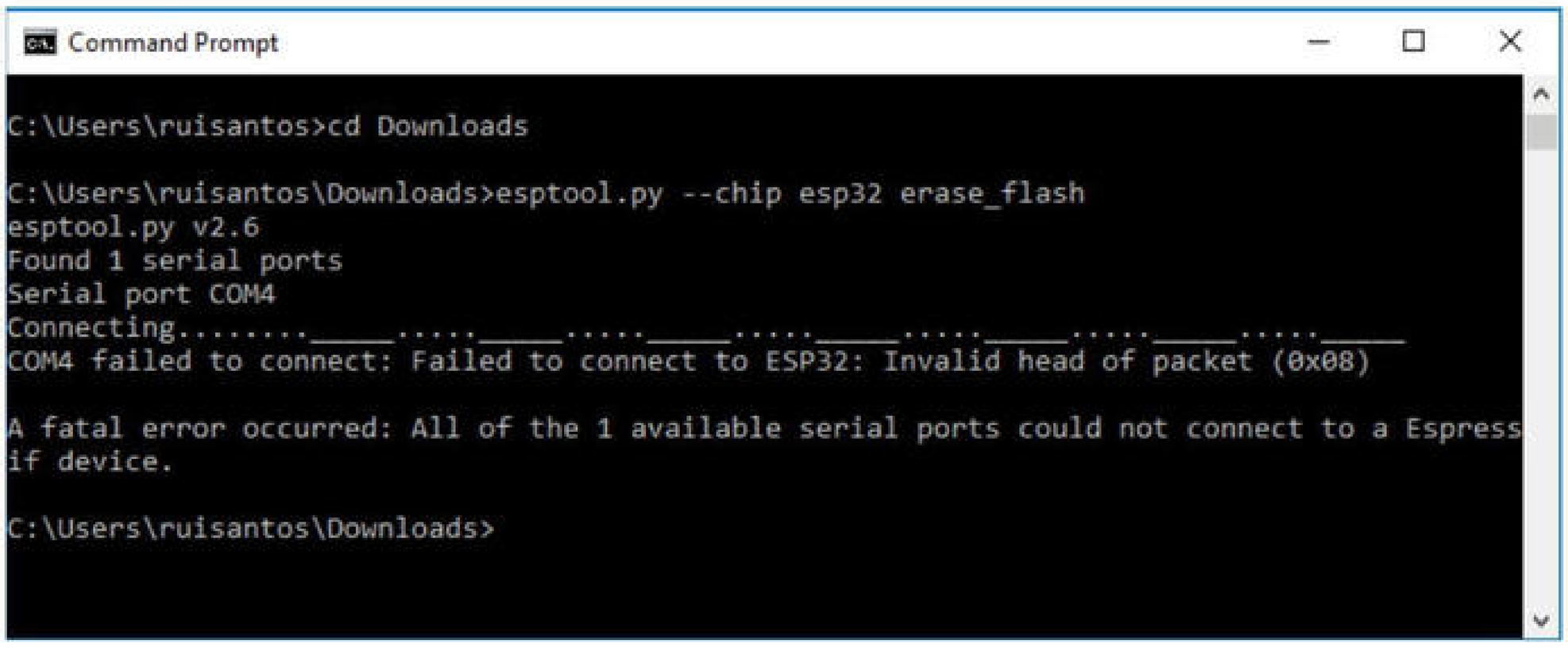
While holding down the "BOOT/FLASH" button, run the following command to erase the ESP32 flash memory:

```
esptool.py --chip esp32 erase_flash
```

When the "Erasing" process begins, you can release the "BOOT/FLASH" button. After a few seconds, the ESP32 flash memory will be erased.



Note: if after the "**Connecting ...**" message you keep seeing new dots appearing, it means that your ESP32 is not in flashing mode. You need to repeat all the steps described earlier and hold the "BOOT/FLASH" button again to ensure that your ESP32 goes into flashing mode and completes the erasing process successfully.



Flashing MicroPython Firmware on ESP32 with esptool.py

With your ESP32 flash memory erased, you can finally flash the MicroPython firmware. You need your serial port name (COM7 in our case) and the ESP32 *.bin* file location. Replace the next command with your details:

```
esptool.py --chip esp32 --port <serial_port> write_flash -z
0x1000 <esp32-X.bin>
```

In our case, the final command looks like this:

```
esptool.py --chip esp32 --port COM7 write_flash -z 0x1000 esp32-
20190113-v1.9.4-779-g5064df207.bin
```

Hold down the "BOOT/FLASH", before running the flash command. After a few seconds this is what you should see:

```
Command Prompt

C:\Users\ruisantos\Downloads>esptool.py --chip esp32 --port COM7 write_flash -z 0x1000 esp32-20190113-v1.9.4-779-g5064df207.bin
esptool.py v2.6
Serial port COM7
Connecting........_
Chip is ESP32D0WDQ6 (revision 1)
Features: WiFi, BT, Dual Core, 240MHz, VRef calibration in efuse, Coding Scheme None
MAC: b4:e6:2d:97:ee:f1
Uploading stub...
Running stub...
Stub running...
Configuring flash size...
Auto-detected Flash size: 4MB
Compressed 1084080 bytes to 685393...
Wrote 1084080 bytes (685393 compressed) at 0x00001000 in 61.5 seconds (effective 141.0 kbit/s)...
Hash of data verified.

Leaving...
Hard resetting via RTS pin...

C:\Users\ruisantos\Downloads>
```

Your ESP32 was successfully flashed with MicroPython firmware!

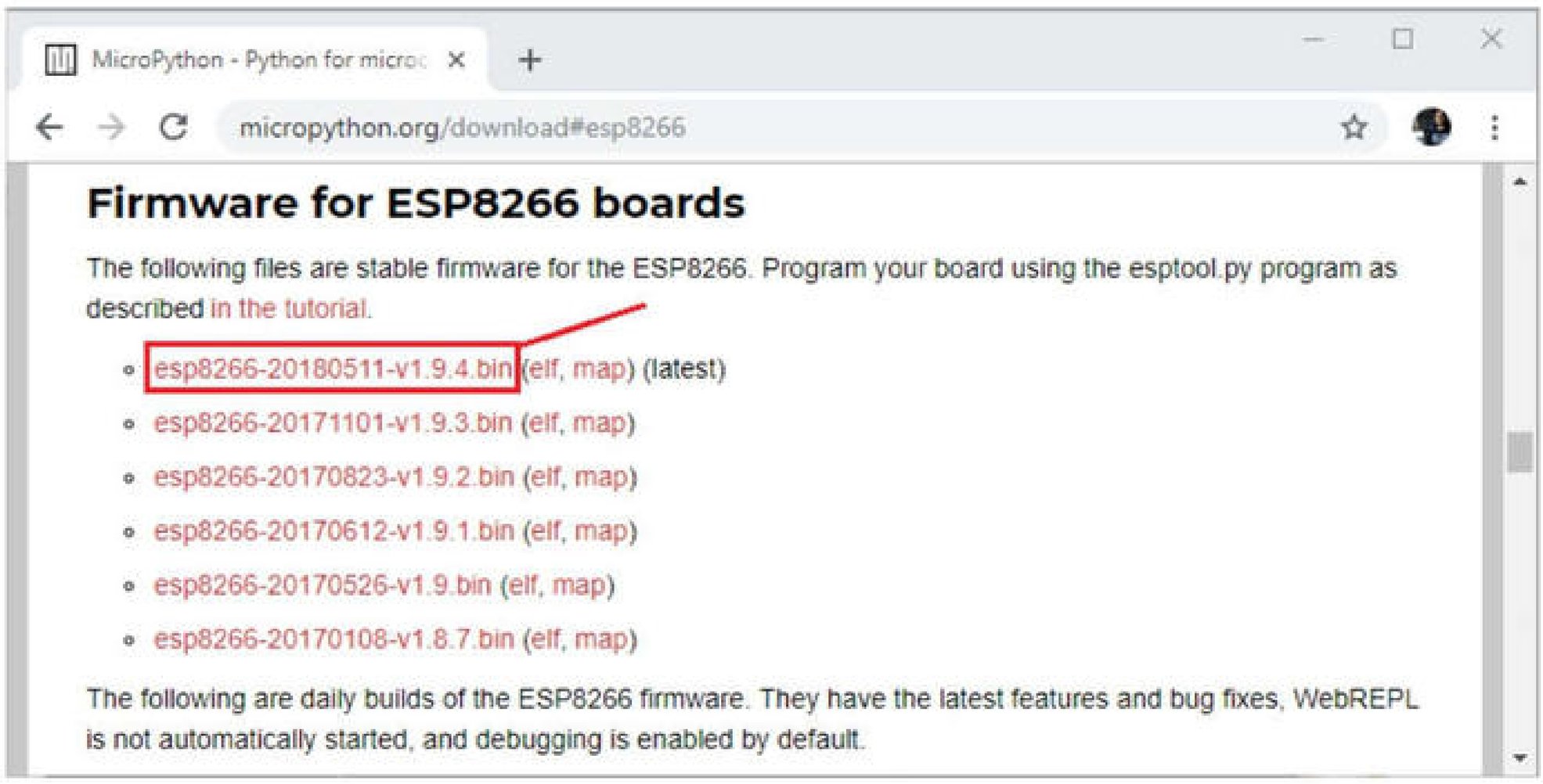
If you receive an error trying to flash the firmware, run the command again and make sure you are holding down the ESP32 "BOOT/FLASH" button.

```
Command Prompt

C:\Users\ruisantos\Downloads>esptool.py --chip esp32 --port COM7 write_flash -z 0x1000 esp32-20190113-v1.9.4-779-g5064df207.bin
esptool.py v2.6
Serial port COM4
Connecting.....____.....____.....____.....____.....____.....____.....____

A fatal error occurred: Failed to connect to ESP32: Invalid head of packet (0x08)

C:\Users\ruisantos\Downloads>
```

[Part 2 – ESP8266] Downloading and Flashing the MicroPython Firmware on ESP8266

To download the latest version of MicroPython firmware for the ESP8266, go to the MicroPython Downloads page and scroll all the way down to the ESP8266 section.

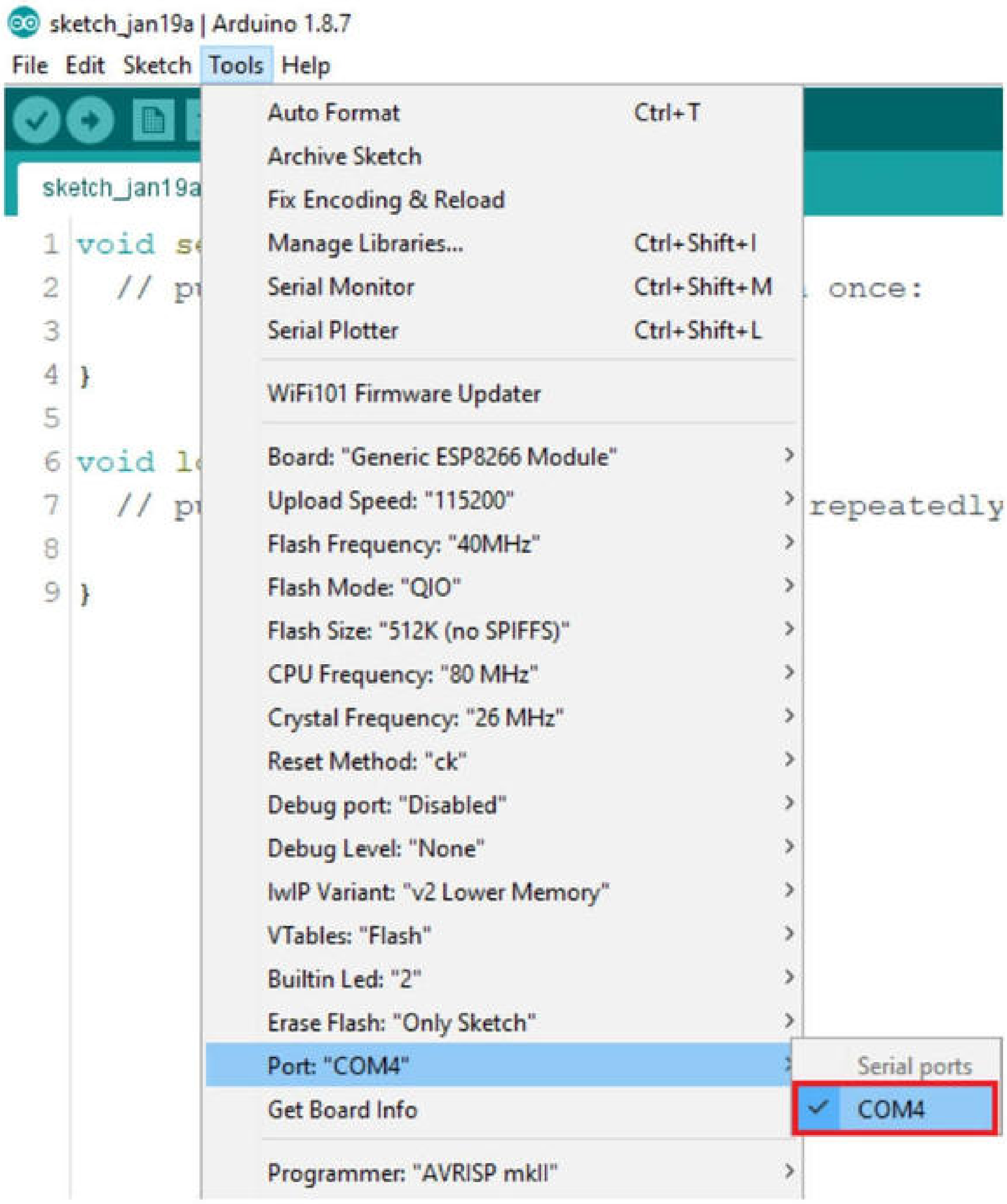
You should see a similar web page (see figure below) with the latest link to download the ESP8266 *.bin* file – for example: *esp8266-20180511-v1.9.4.bin*.



Finding the Serial Port Name

It's a bit different to find the Serial port name in each operating system, so for simplicity reasons we recommend finding your ESP serial port name through the Arduino IDE. Follow these steps:

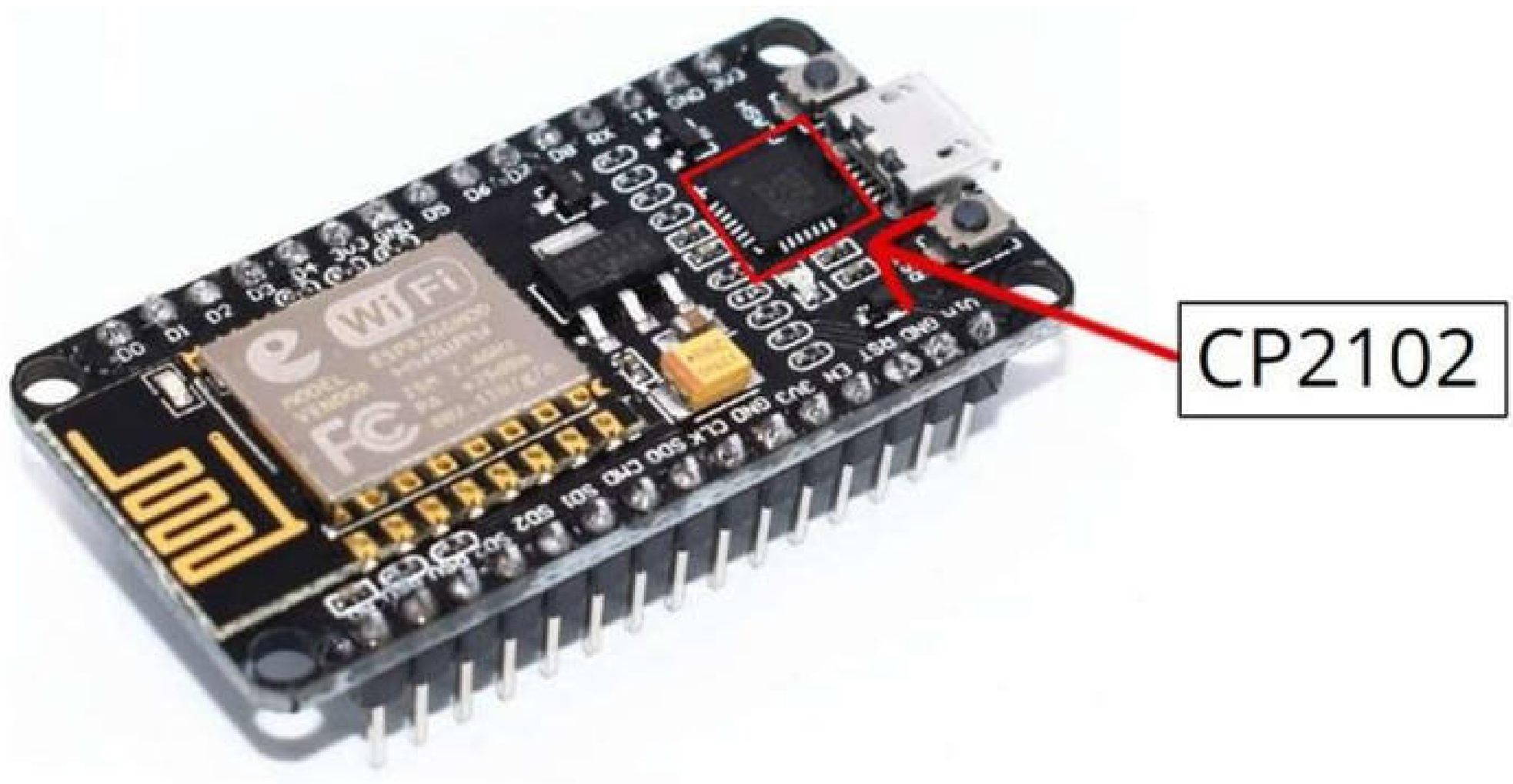
1. Connect your board to your computer
2. Open the Arduino IDE
3. Go to **Tools** ▸ **Port**
4. Save your ESP8266 serial port name (in our case it's COM4)
5. Close your Arduino IDE software



Important: if you plug your ESP8266 board to your computer, but you can't find the ESP8266 Port available in your Arduino, it might be one of these two problems: **1.** USB drivers missing or **2.** USB cable without data wires.

1. If you don't see your ESP's COM port available, this often means you don't have the USB drivers installed. Take a closer look at the chip next to the voltage regulator on board and check its name.

The ESP8266 ESP-12E NodeMCU board uses the CP2102 chip.



Go to Google and search for your specific chip to find the drivers and install them in your operating system.

You can download the CP2102 drivers on the Silicon Labs website.

After they are installed, restart the Arduino IDE and you should see the serial port in the **Tools ▸ Port** menu.

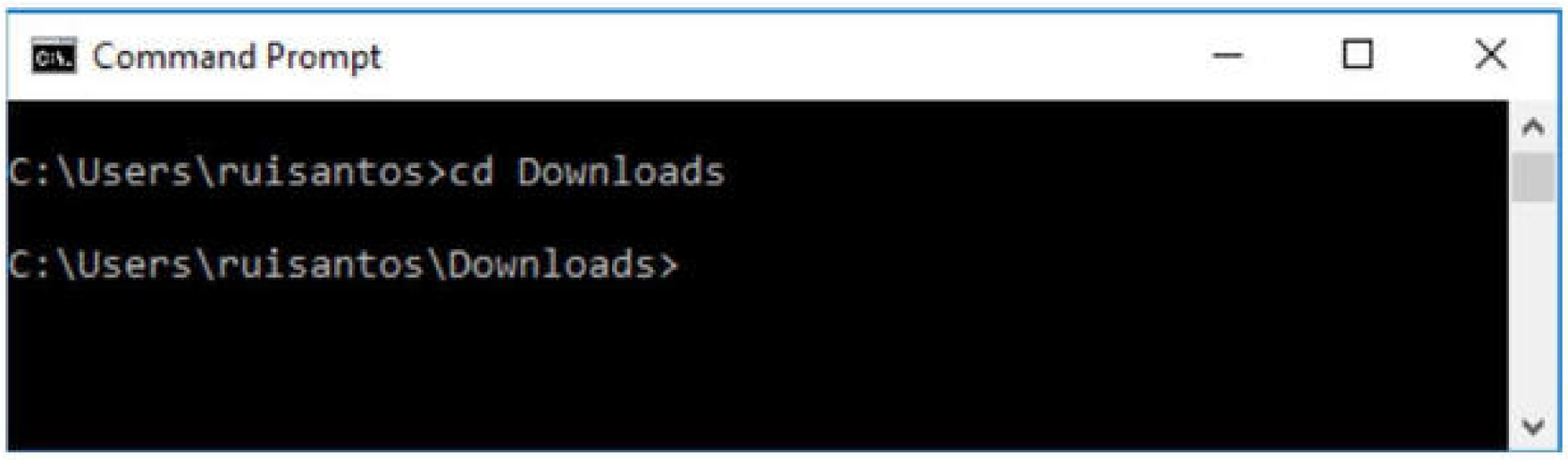
2. If you have the drivers installed, but you can't see your device, double-check that you're using a USB cable with data wires.

USB cables from powerbanks often don't have data wires (they are charge only). So, your computer will never establish a serial communication with your ESP8266. Using a proper USB cable should solve your problem.

Finding your MicroPython .bin file

After downloading the ESP8266 *.bin* file, it should be in your **Downloads** folder. So, with your Terminal window, you'll need to navigate to the **Downloads** folder using the `cd` command:

```
cd Downloads
```



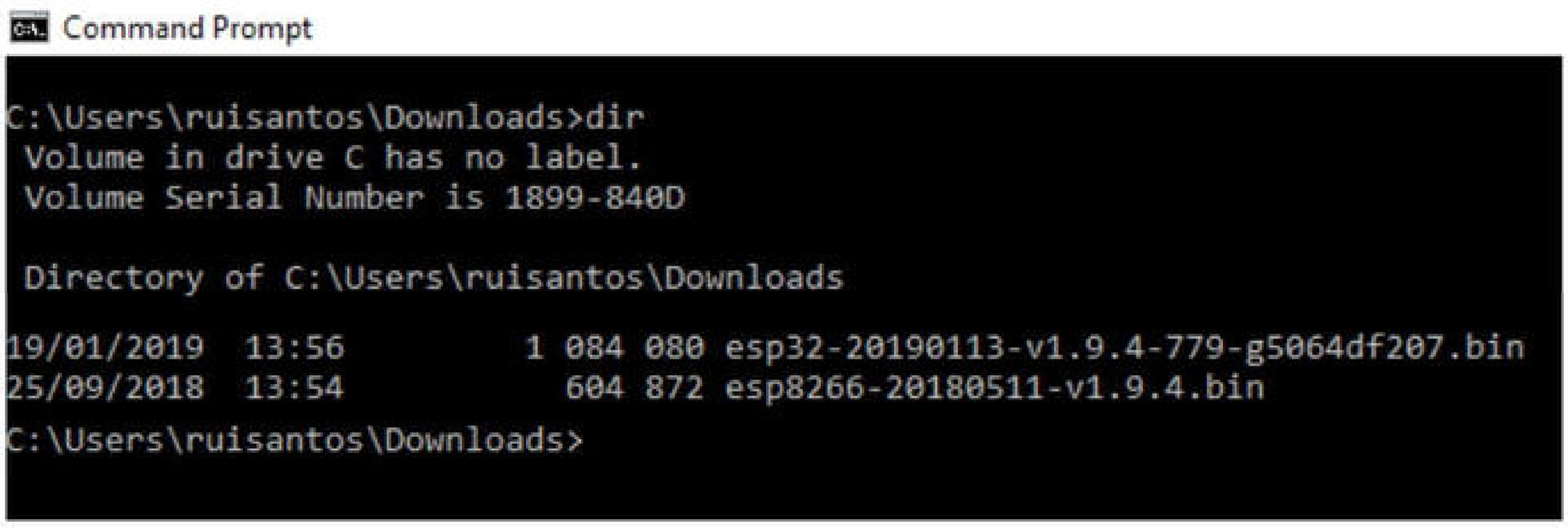
List all files in your **Downloads** folder to ensure that's where the *.bin* file is located.

In Windows, you use:

```
dir
```

On Mac OS X or Linux, run the next command:

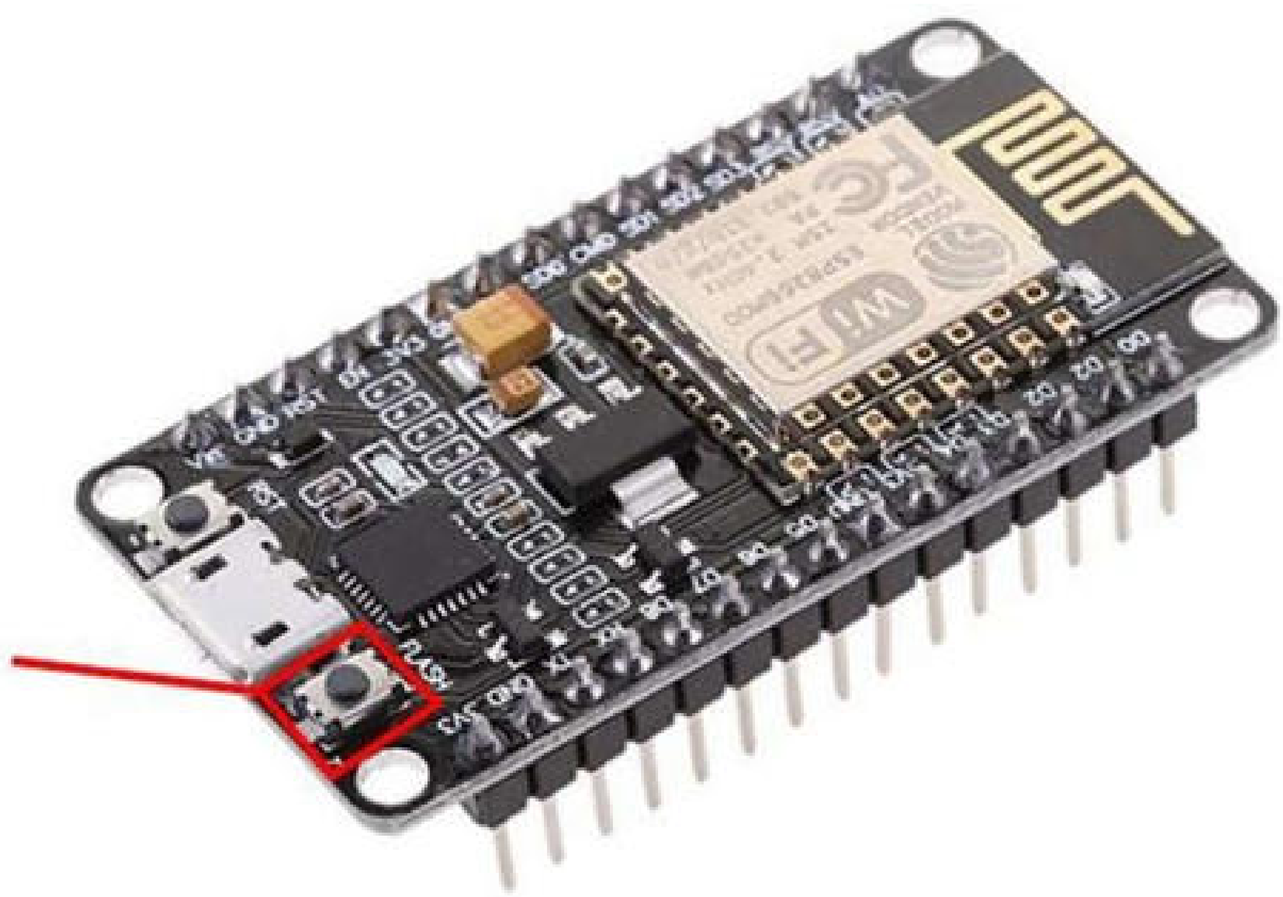
```
ls
```



As you can see in the preceding screenshot, the ESP8266 *.bin* file is located in the **Downloads** folder: *esp8266-20180511-v1.9.4.bin*.

Erasing ESP8266 Flash Memory

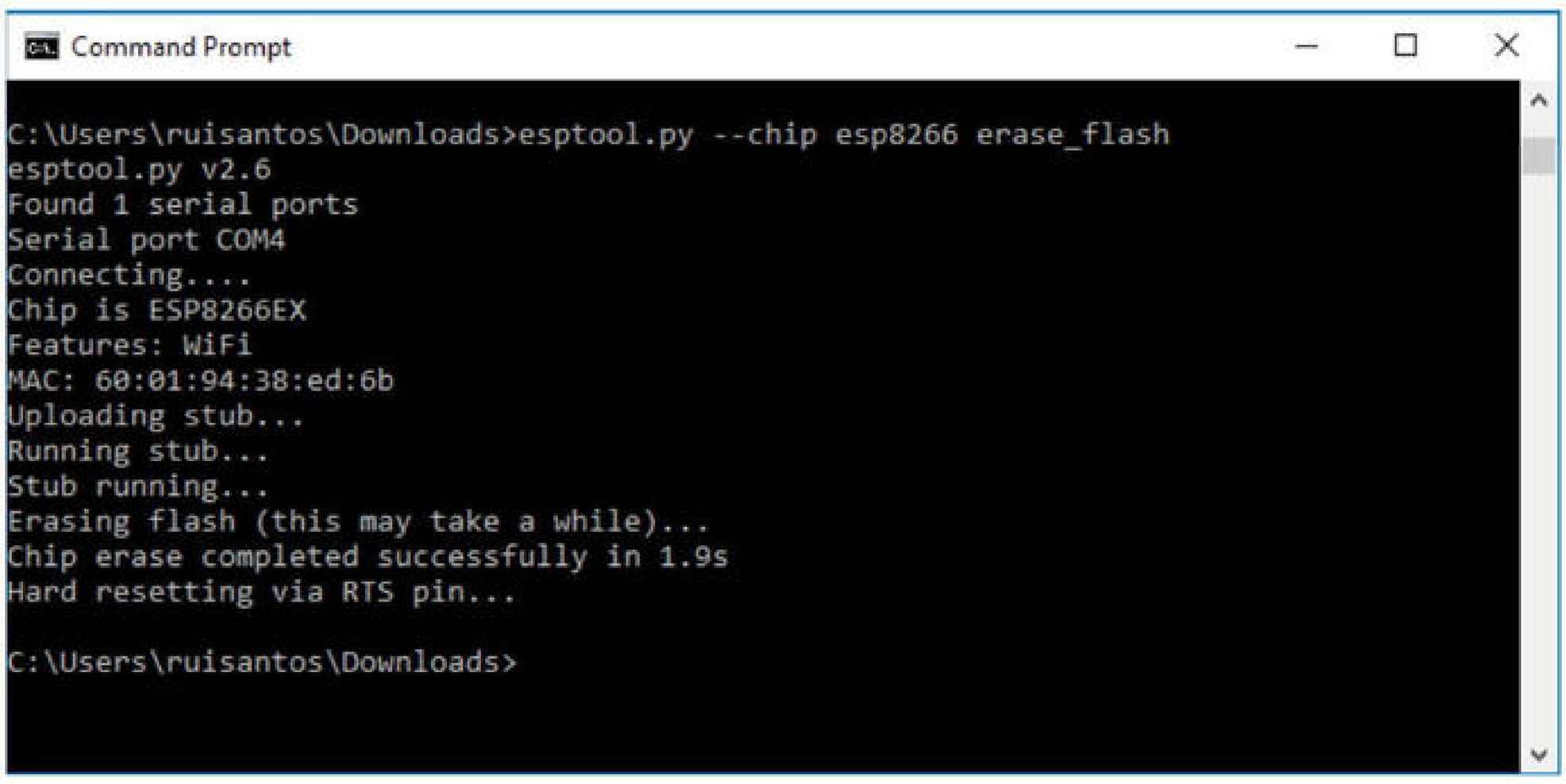
Before flashing the MicroPython firmware, you need to erase the ESP8266 flash memory. So, with your ESP8266 connected to your computer, hold-down the "BOOT/FLASH" button in your ESP8266 board:



While holding down the "BOOT/FLASH" button, run the following command to erase the ESP8266 flash memory:

```
esptool.py --chip esp8266 erase_flash
```

When the "Erasing" process begins, you can release the "BOOT/FLASH" button. After a few seconds, the ESP8266 flash memory will be erased.



```
Command Prompt

C:\Users\ruisantos\Downloads>esptool.py --chip esp8266 erase_flash
esptool.py v2.6
Found 1 serial ports
Serial port COM4
Connecting....
Chip is ESP8266EX
Features: WiFi
MAC: 60:01:94:38:ed:6b
Uploading stub...
Running stub...
Stub running...
Erasing flash (this may take a while)...
Chip erase completed successfully in 1.9s
Hard resetting via RTS pin...

C:\Users\ruisantos\Downloads>
```

Note: if after the "**Connecting ...**" message you keep seeing new dots appearing, it means that your ESP8266 is not in flashing mode. You need to repeat all the steps described earlier and hold the "BOOT/FLASH" button again to ensure that your ESP8266 goes into flashing mode and completes the erasing process successfully.

Flashing MicroPython Firmware on ESP8266 with esptool.py

With your ESP8266 flash memory erased, you can finally flash the MicroPython firmware. You need your serial port name (COM7 in our case) and the ESP8266 *.bin* file location. Replace the next command with your details:

```
esptool.py --chip esp8266 --port <serial_port> write_flash --
flash_mode dio --flash_size detect 0x0 <esp8266-X.bin>
```

In our case, the final command looks like this:

```
esptool.py --chip esp8266 --port COM4 write_flash --flash_mode
dio --flash_size detect 0x0 esp8266-20180511-v1.9.4.bin
```

Hold down the "BOOT/FLASH", before running the flash command. After a few seconds this is what you should see:

```
Command Prompt

C:\Users\ruisantos\Downloads>esptool.py --chip esp8266 --port COM4 write_flash --flash_mode dio --flash_size detect
esptool.py v2.6
Serial port COM4
Connecting....
Chip is ESP8266EX
Features: WiFi
MAC: 60:01:94:38:ed:6b
Uploading stub...
Running stub...
Stub running...
Configuring flash size...
Auto-detected Flash size: 4MB
Flash params set to 0x0240
Compressed 604872 bytes to 394893...
Wrote 604872 bytes (394893 compressed) at 0x00000000 in 35.3 seconds (effective 137.1 kbit/s)...
Hash of data verified.

Leaving...
Hard resetting via RTS pin...

C:\Users\ruisantos\Downloads>
```

Your ESP8266 was successfully flashed with MicroPython firmware!

Note: if you receive an error trying to flash the firmware, run the command again and make sure you are holding down the ESP8266 "BOOT/FLASH" button.

Installing Thonny IDE

In this guide, we provide instructions to install Thonny IDE in different operating systems, read the section that fits your needs:

A) Installing Thonny IDE – Windows PC

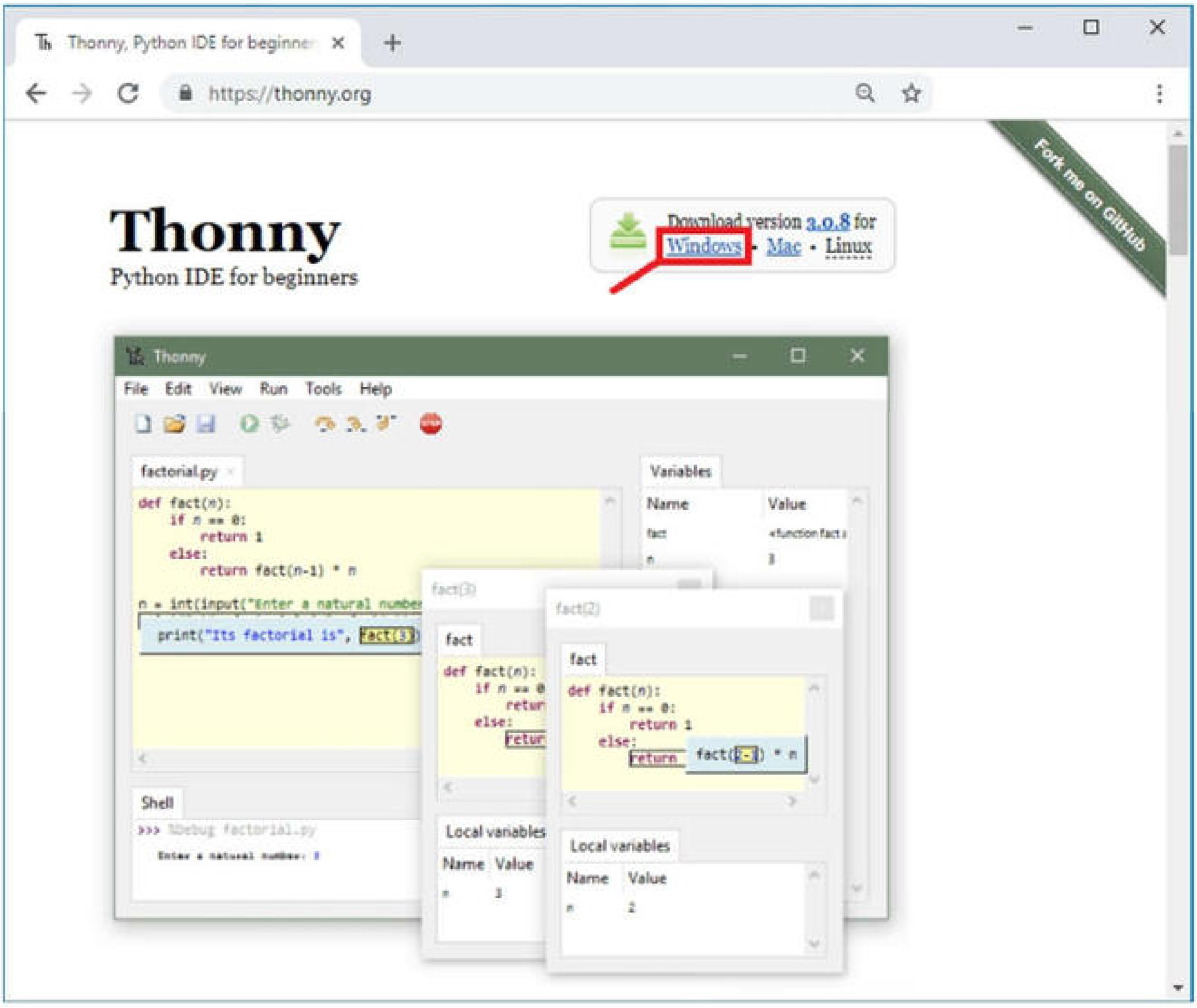
B) Installing Thonny IDE – Mac OS X

C) Installing Thonny IDE – Linux

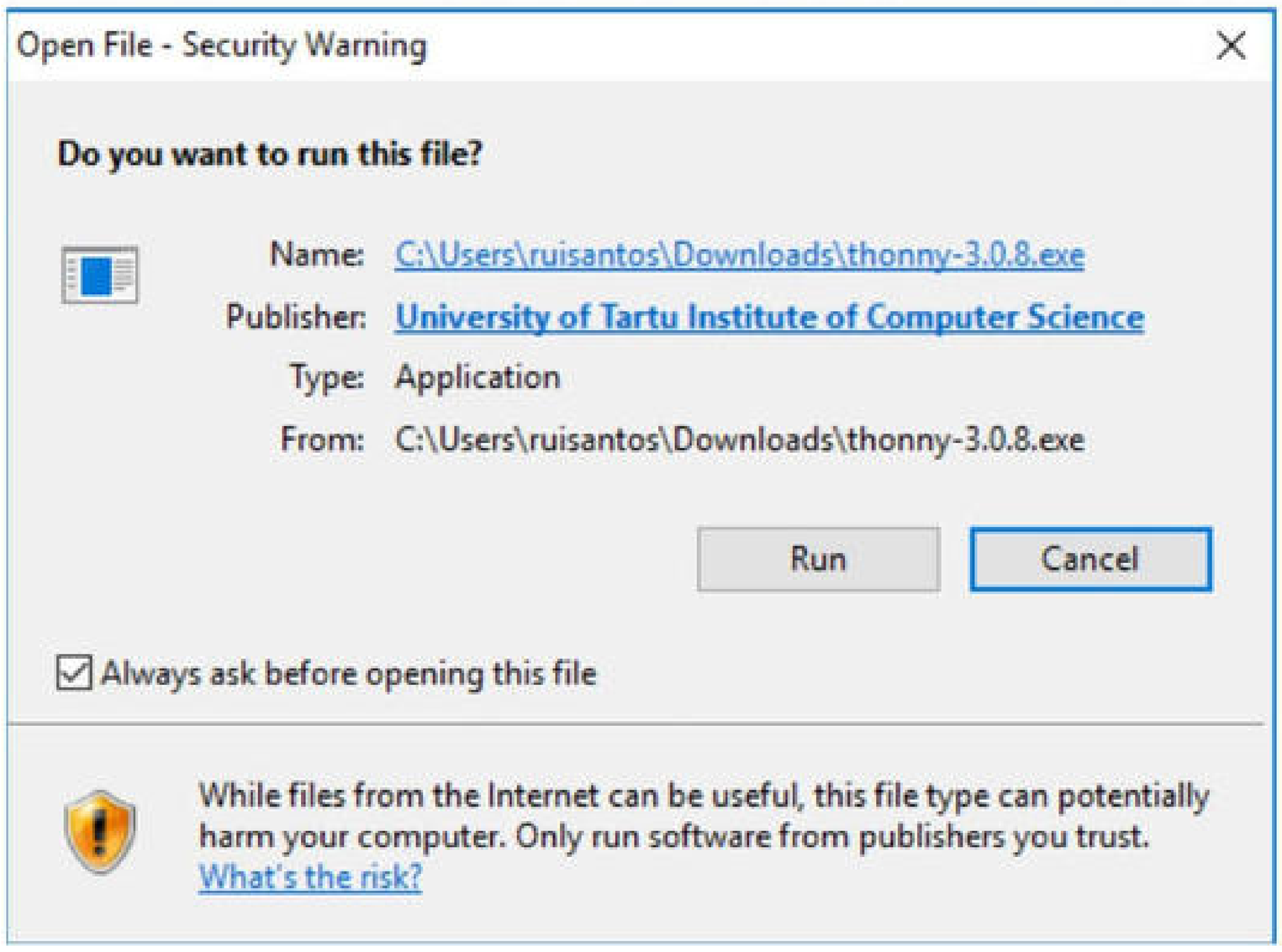
A) Installing Thonny IDE – Windows PC

To install Thonny on your Windows PC, follow the next instructions:

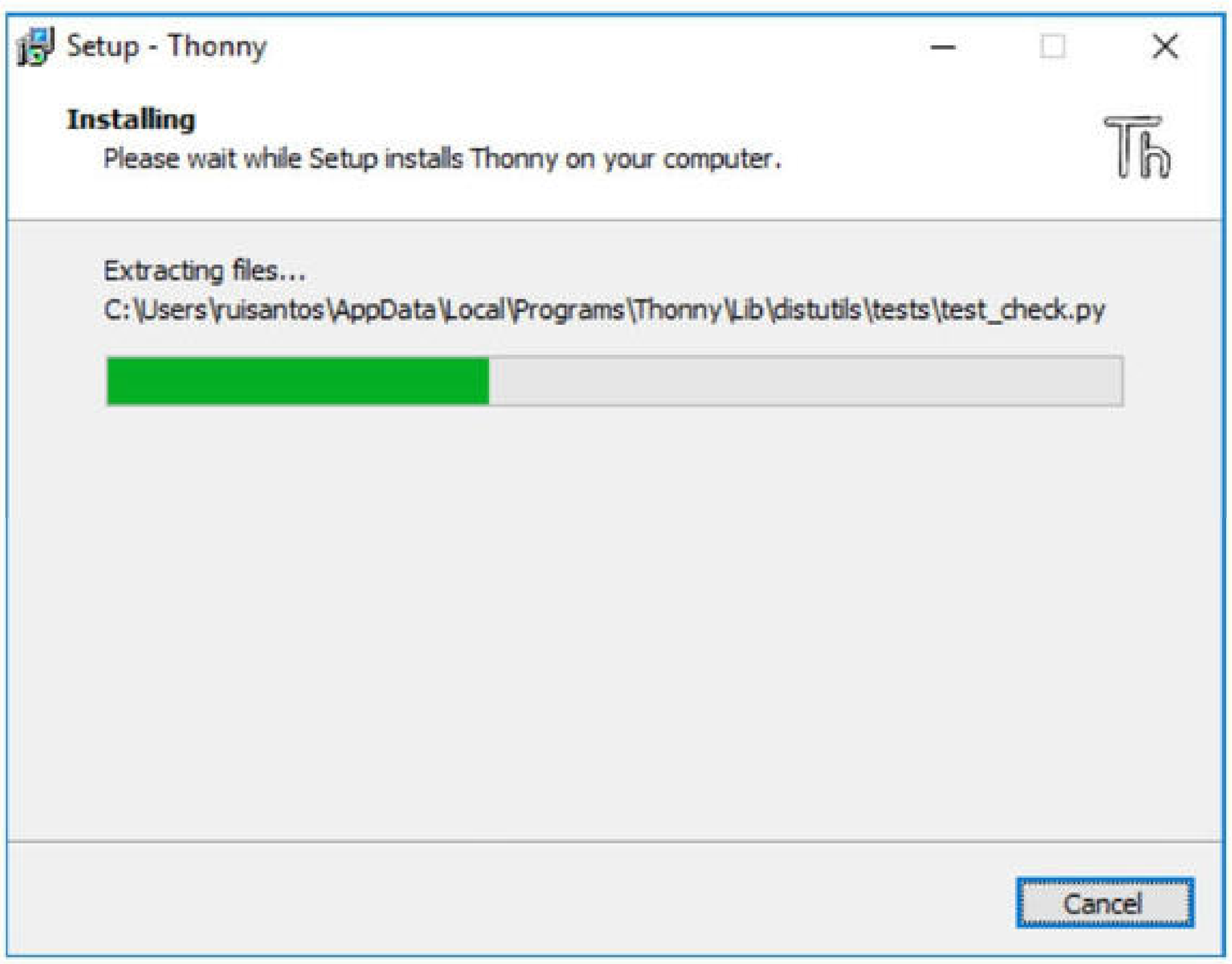
1. Go to https://thonny.org
2. Download the version for Windows and wait a few seconds while it downloads.



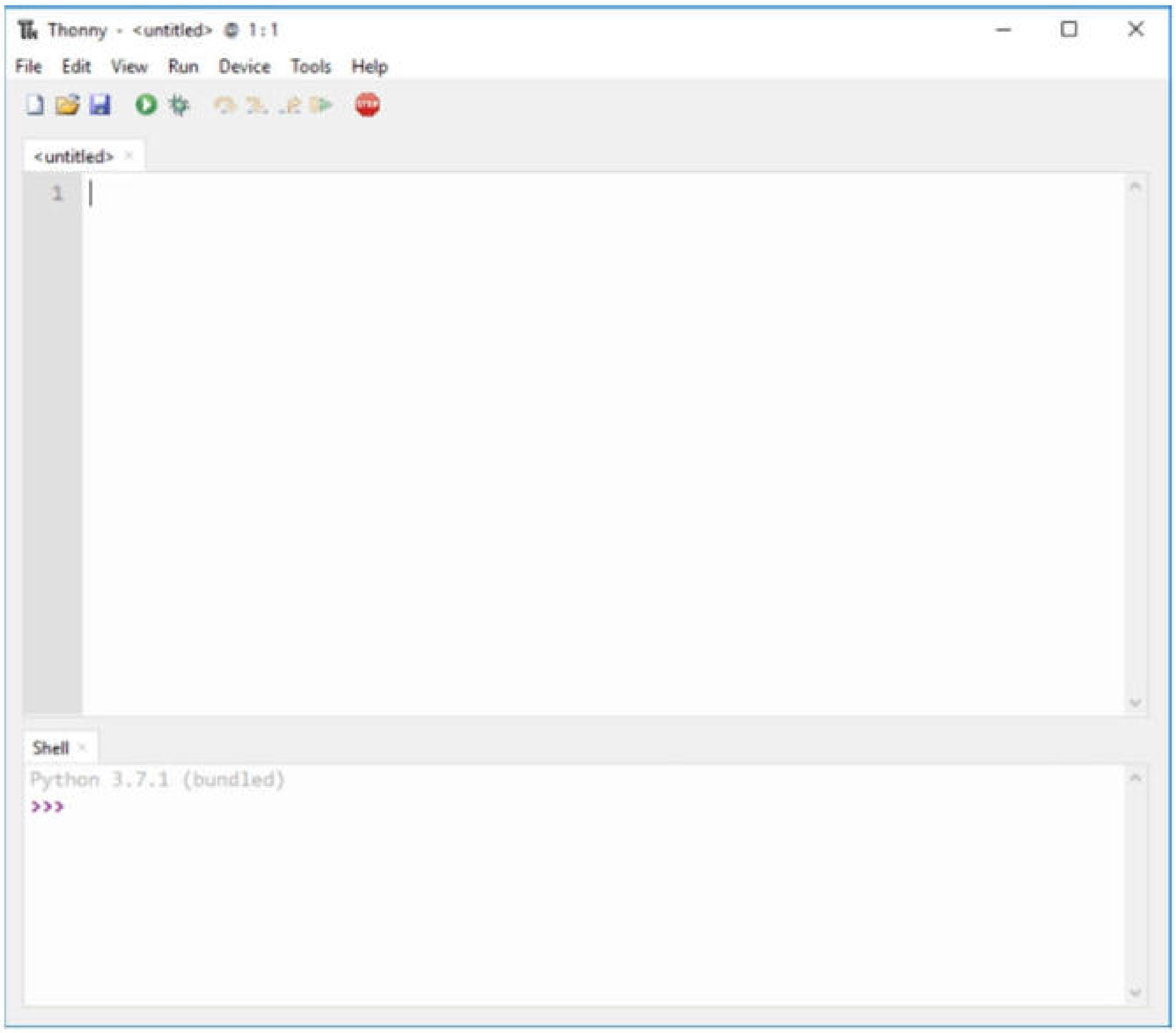
3. Run the .exe file.



4. Follow the installation wizard to complete the installation process. You just need to click "**Next**".



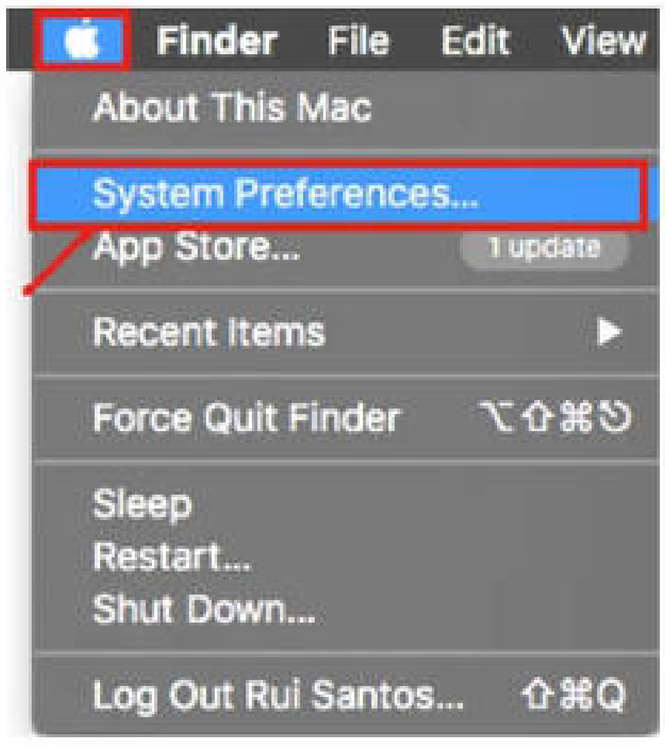
5. After completing the installation, open Thonny IDE. A window as shown in the following figure should open.



B) Installing Thonny IDE – Mac OS X

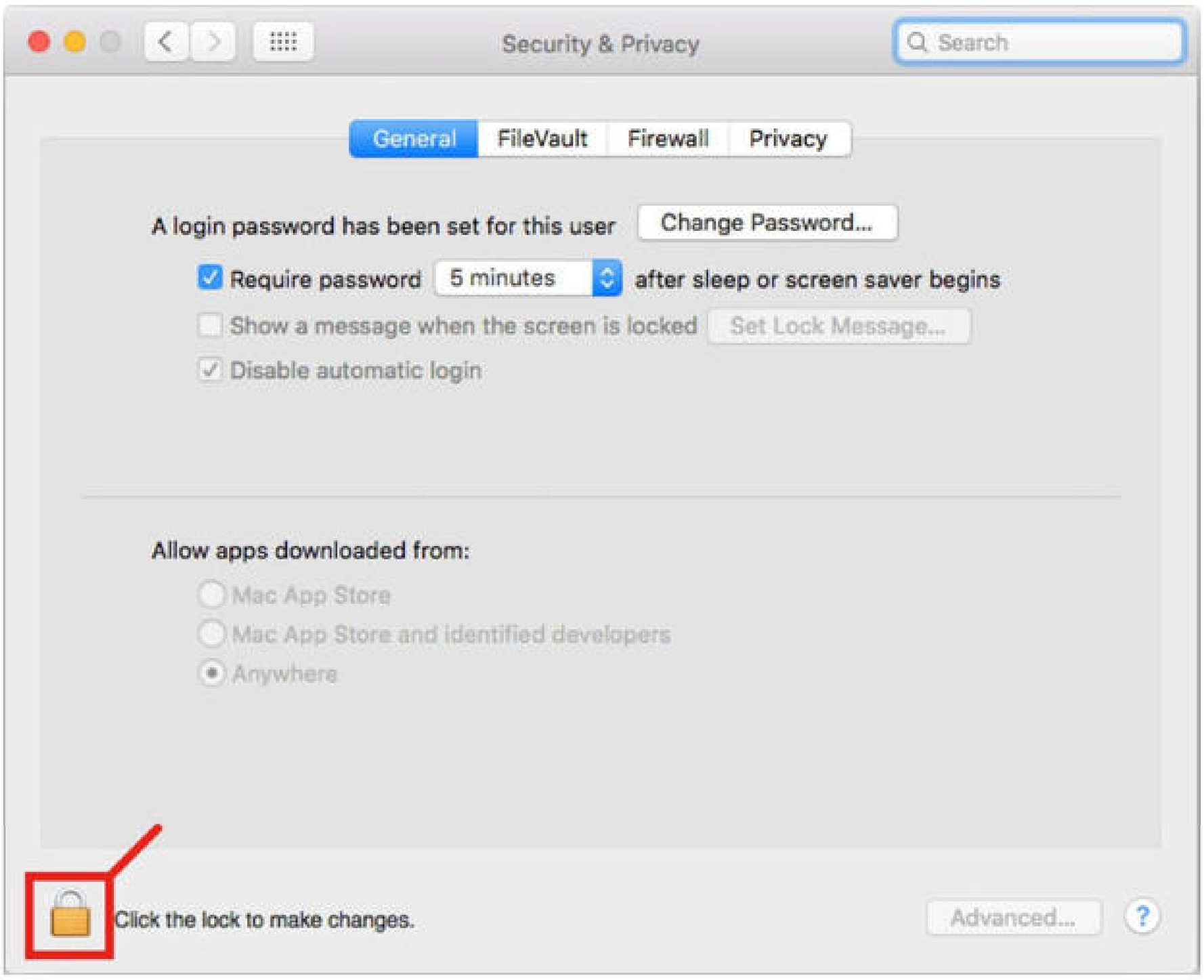
Since Thonny IDE is open source and downloaded from the Internet, it's not a verified app in the App Store. For security reasons, Mac OS X blocks unknown apps to run on your computer. Follow these next instructions to enable any downloaded software to run in your Mac.

1. Open the "**System Preferences...**" menu.



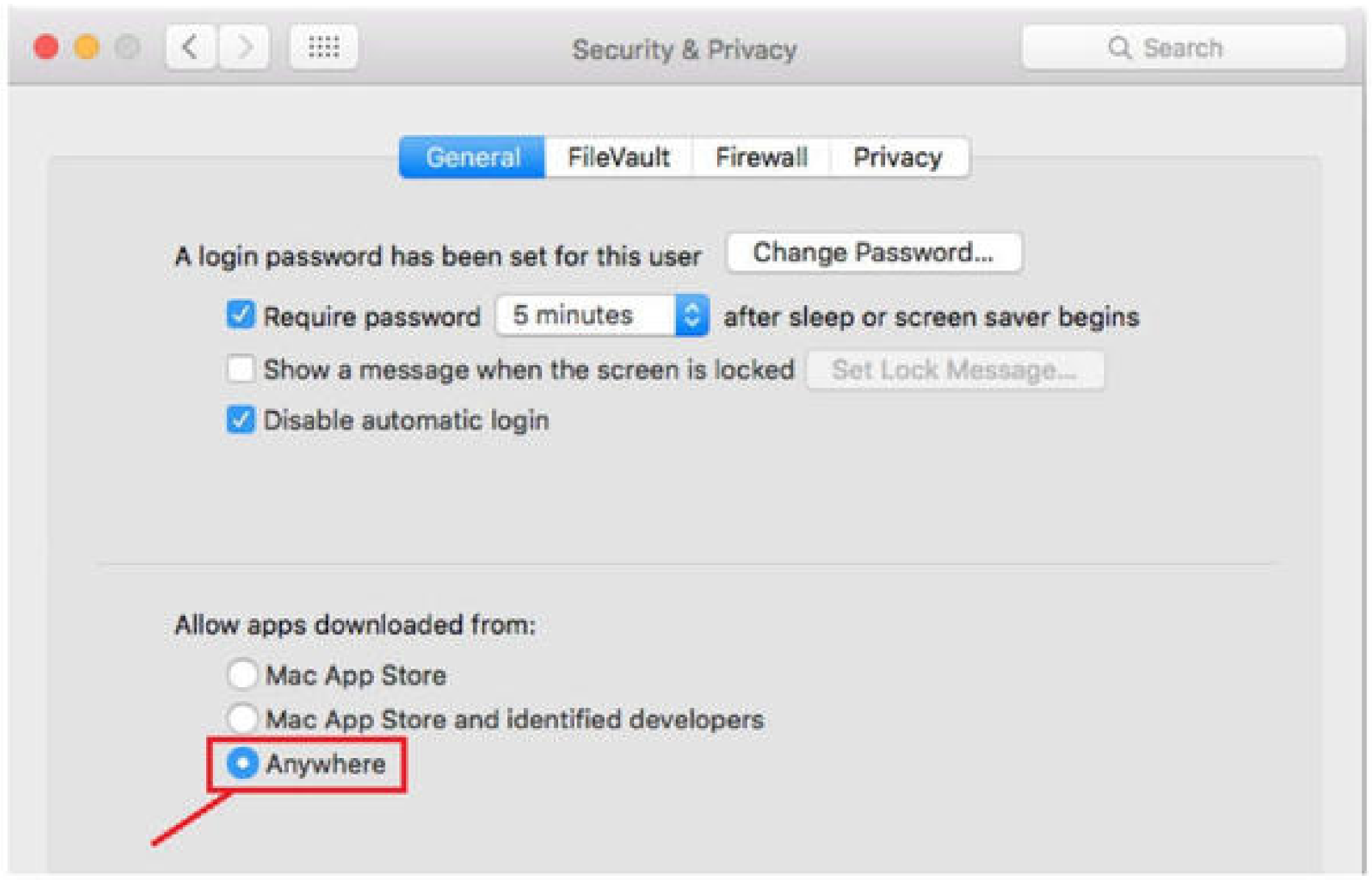
2. Open the "**Security & Privacy**" menu.

3. At the bottom left corner, click the lock icon to modify your "**Security & Privacy**" settings:



4. Type your username/password and click the "**Unlock**" button.

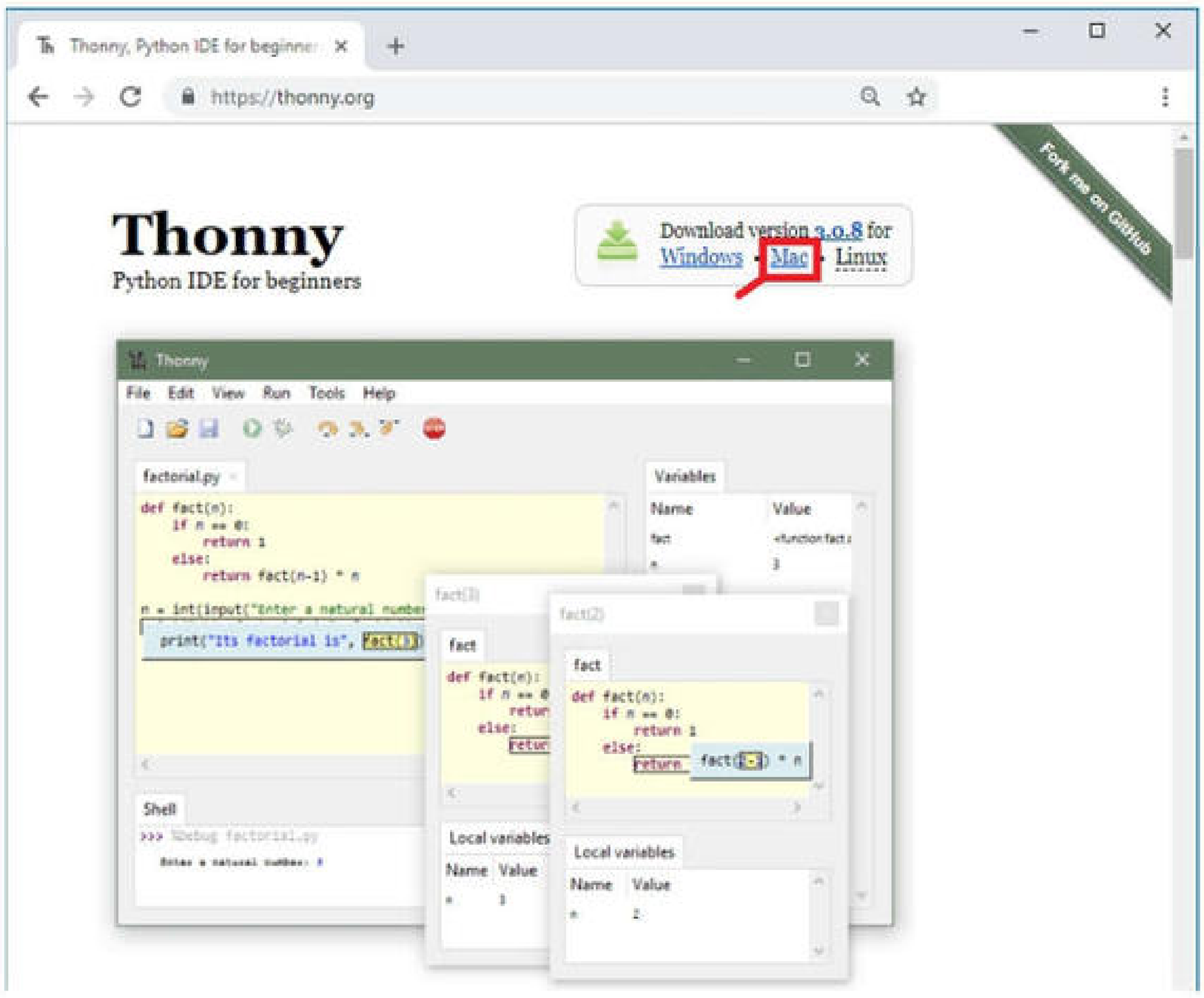
5. Finally, select the option "**Allow apps downloaded from: Anywhere**".



That's it, you can close that window.

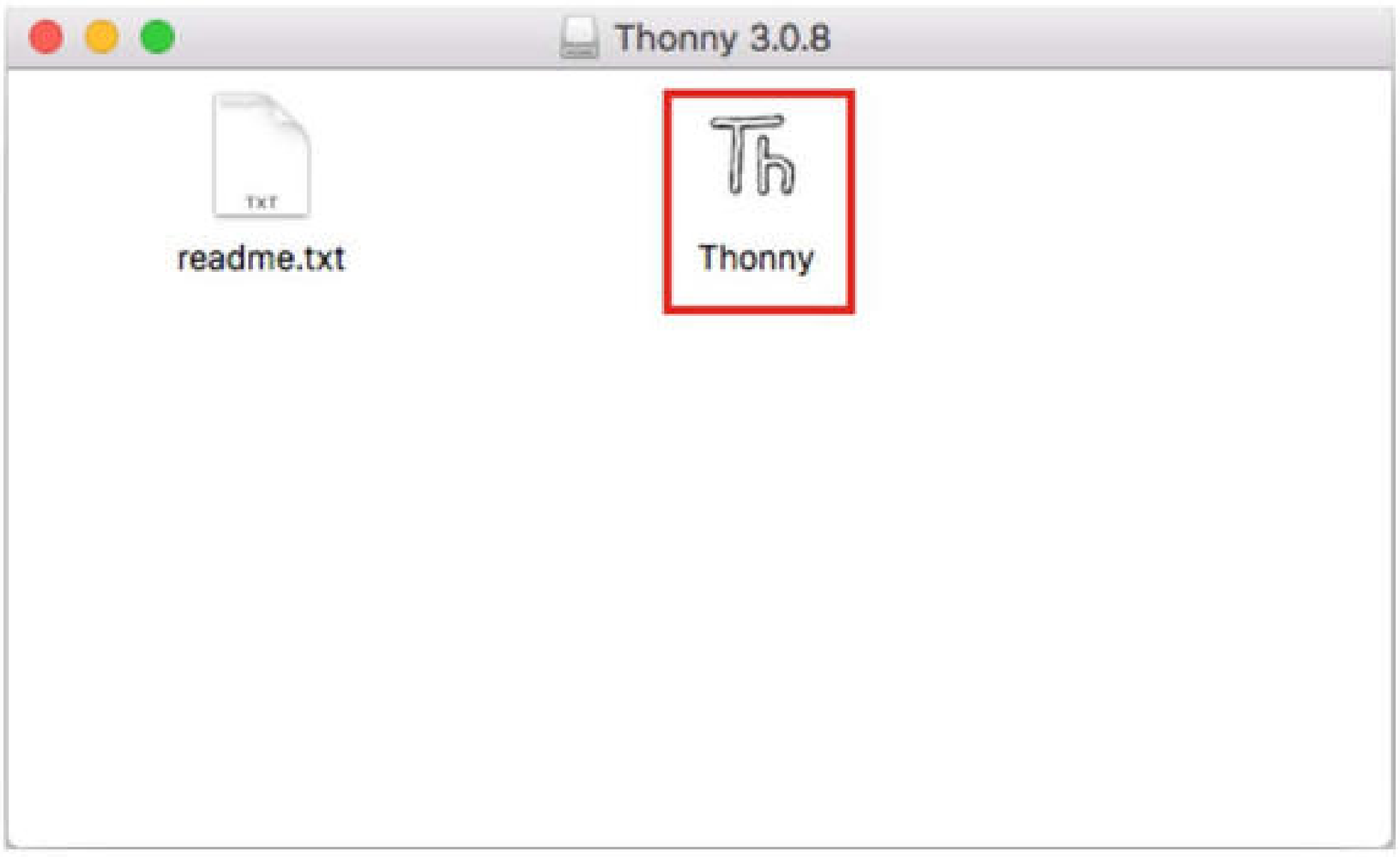
To install Thonny on Mac OS X, follow the next instructions:

1. Go to https://thonny.org
2. Download the version for Mac OS X and wait a few seconds while it downloads.

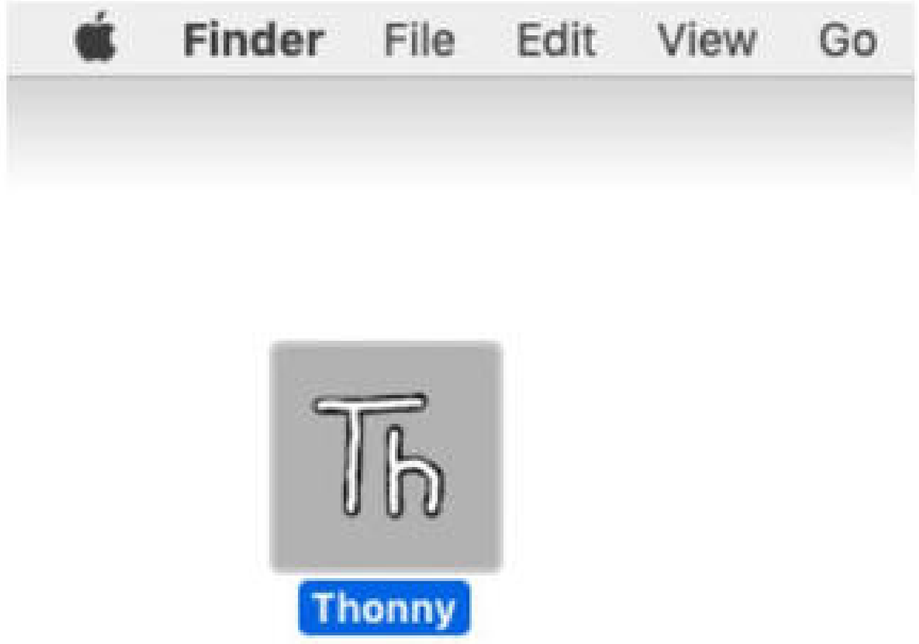


3. Open the *.dmg* file.

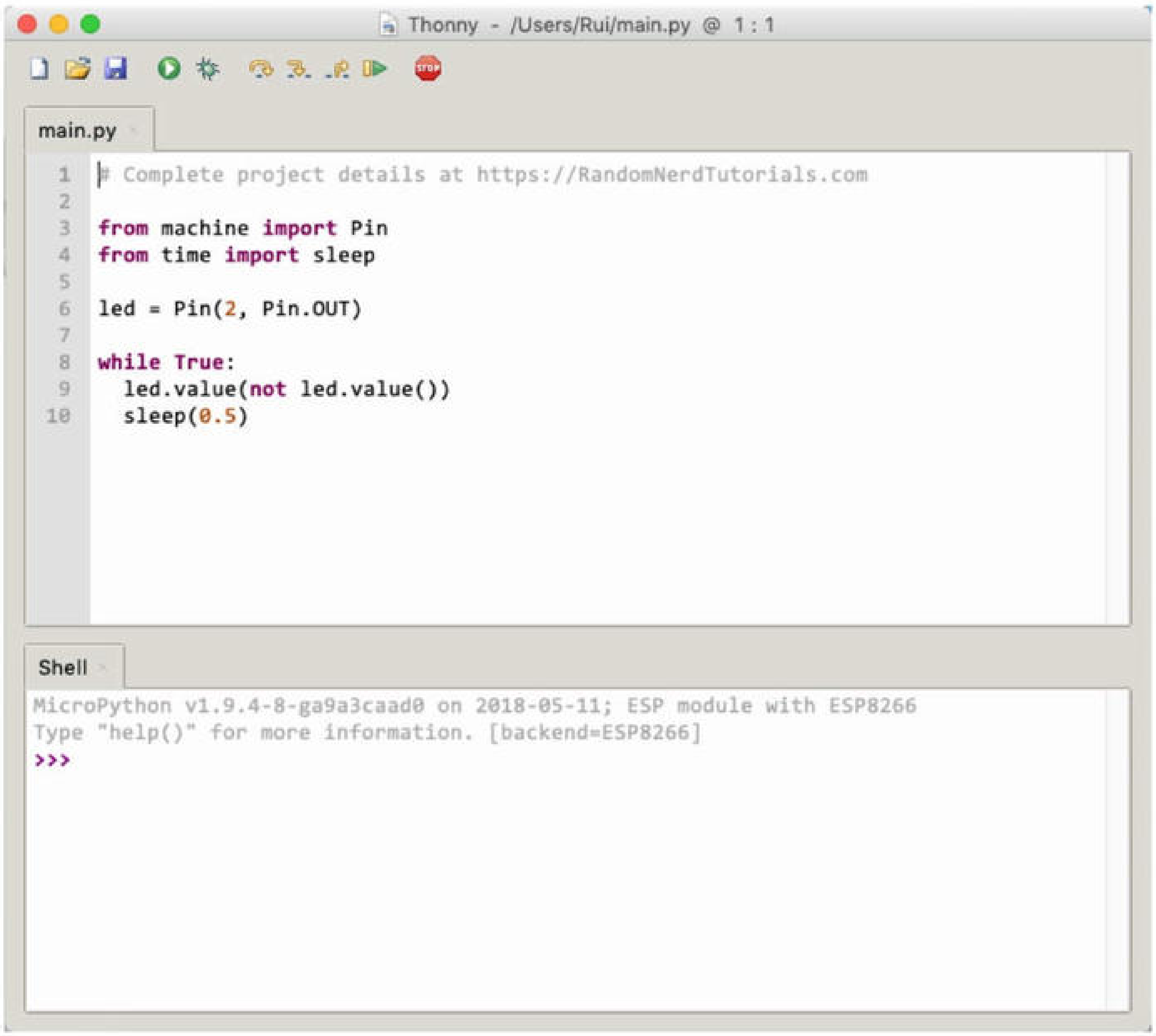
4. Drag the "Thonny" application to your Desktop:



5. Thonny IDE is now installed and you can double-click to open it:



6. After the installation is completed, open Thonny IDE. A window as shown in the following figure should open.



C) Installing Thonny IDE – Linux

To install Thonny on your Linux computer, it depends on your Linux distribution and version, follow the next instructions for your system. First, we recommend installing these dependencies: `python3`, `python3-pip`, and `python3-tk`

If you're in Ubuntu, you can install the Python dependencies like this:

```
sudo apt install python3 python3-pip python3-tk
```

After having Python3, pip3, and Python3 Tkinter, you can install Thonny IDE.

Ubuntu (after running that command, you'll need to press Enter again to install the software):

```
bash <(wget -O - https://thonny.org/installer-for-linux)
```

Or you can install Thonny IDE with pip3:

```
sudo pip3 install thonny
```

Fedora since 27:

```
sudo dnf install thonny
```

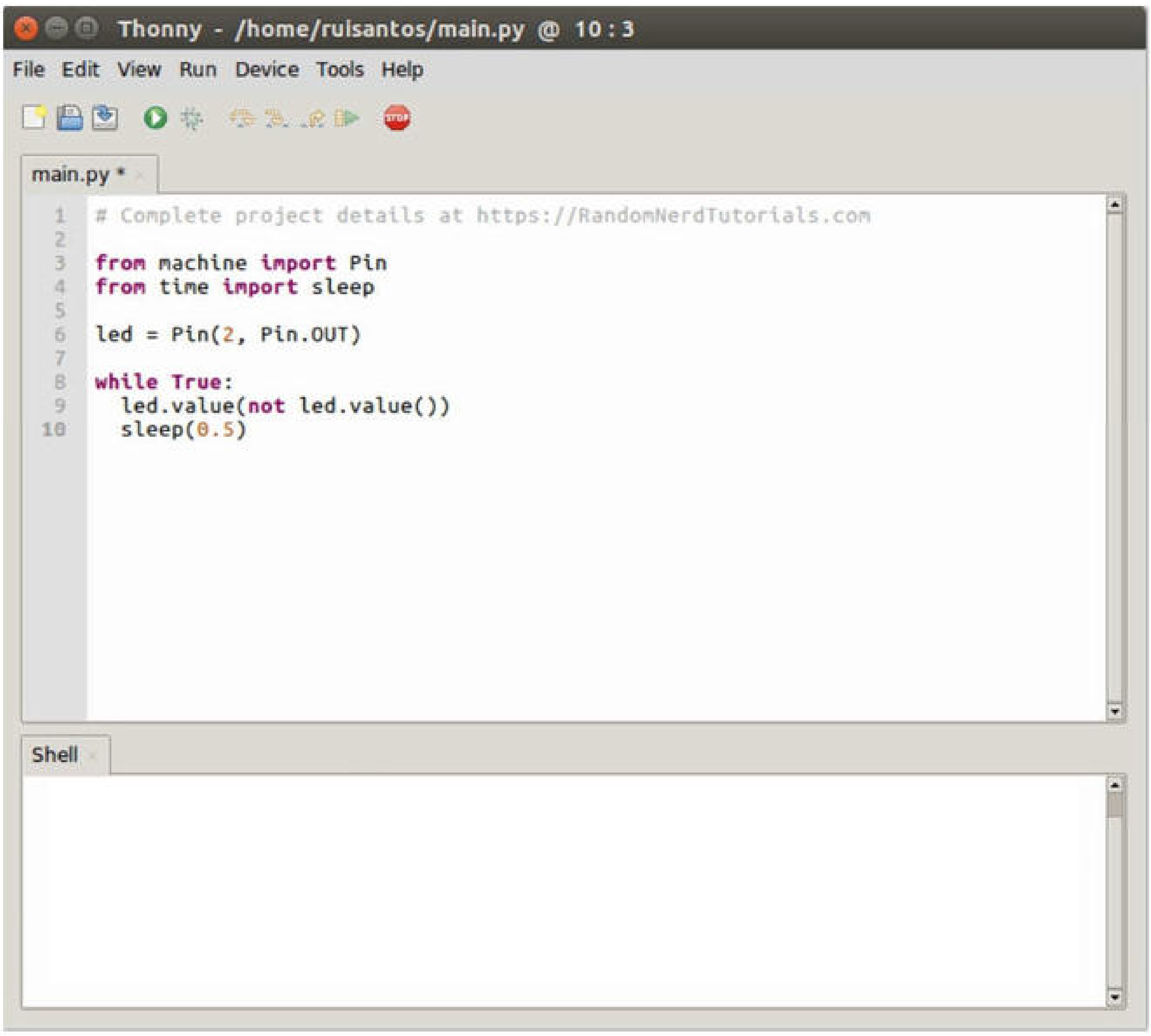
Raspbian since Stretch (installed by default):

```
sudo apt install python3-thonny
```

After installing Thonny IDE and depending on your installation method, to open Thonny IDE, you either need go to the search bar and type "Thonny" to find it, or if you installed Thonny IDE using pip3, you can type in your terminal window:

```
thonny
```

Then, it should open Thonny IDE:

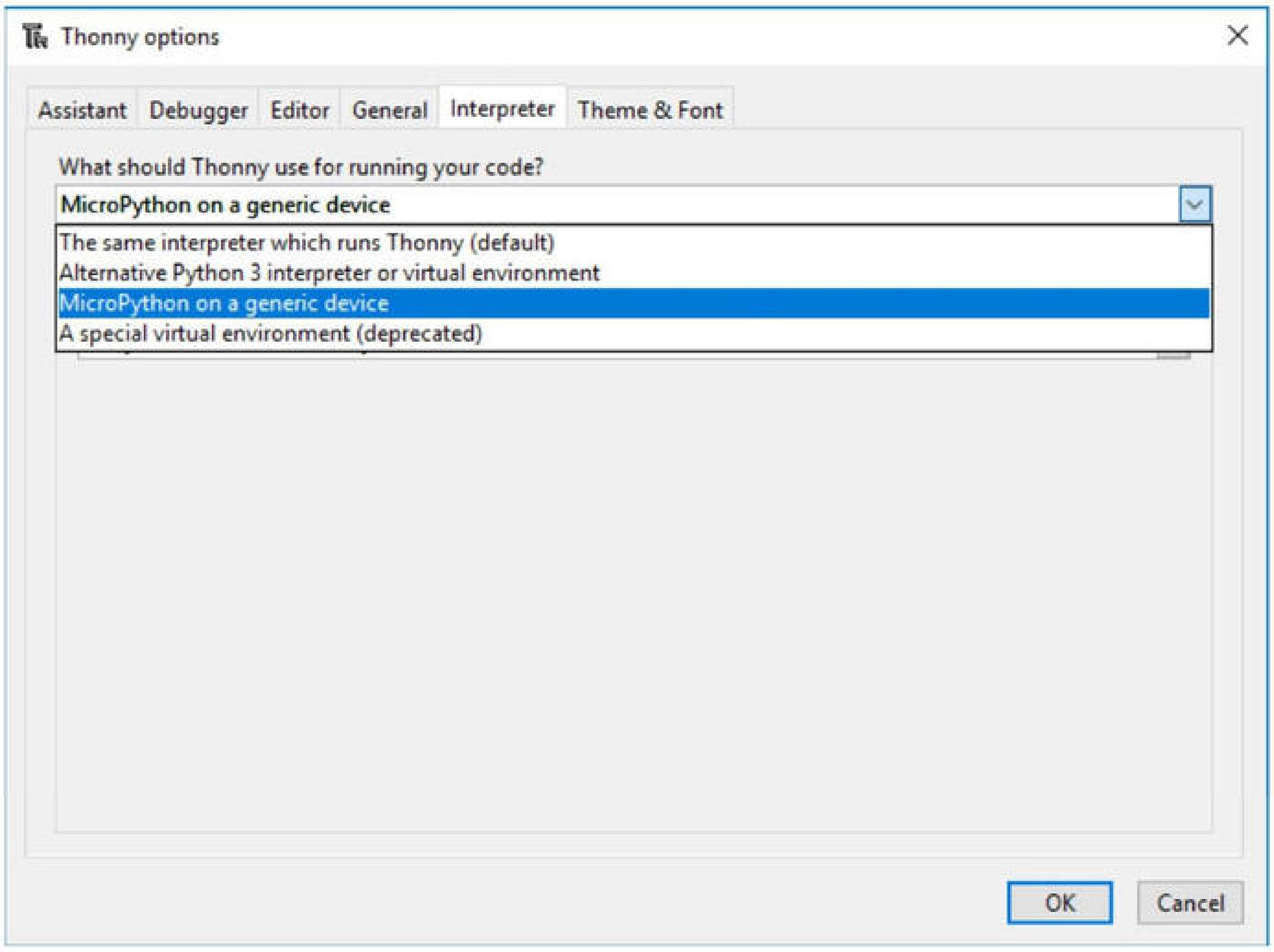


Testing the Installation

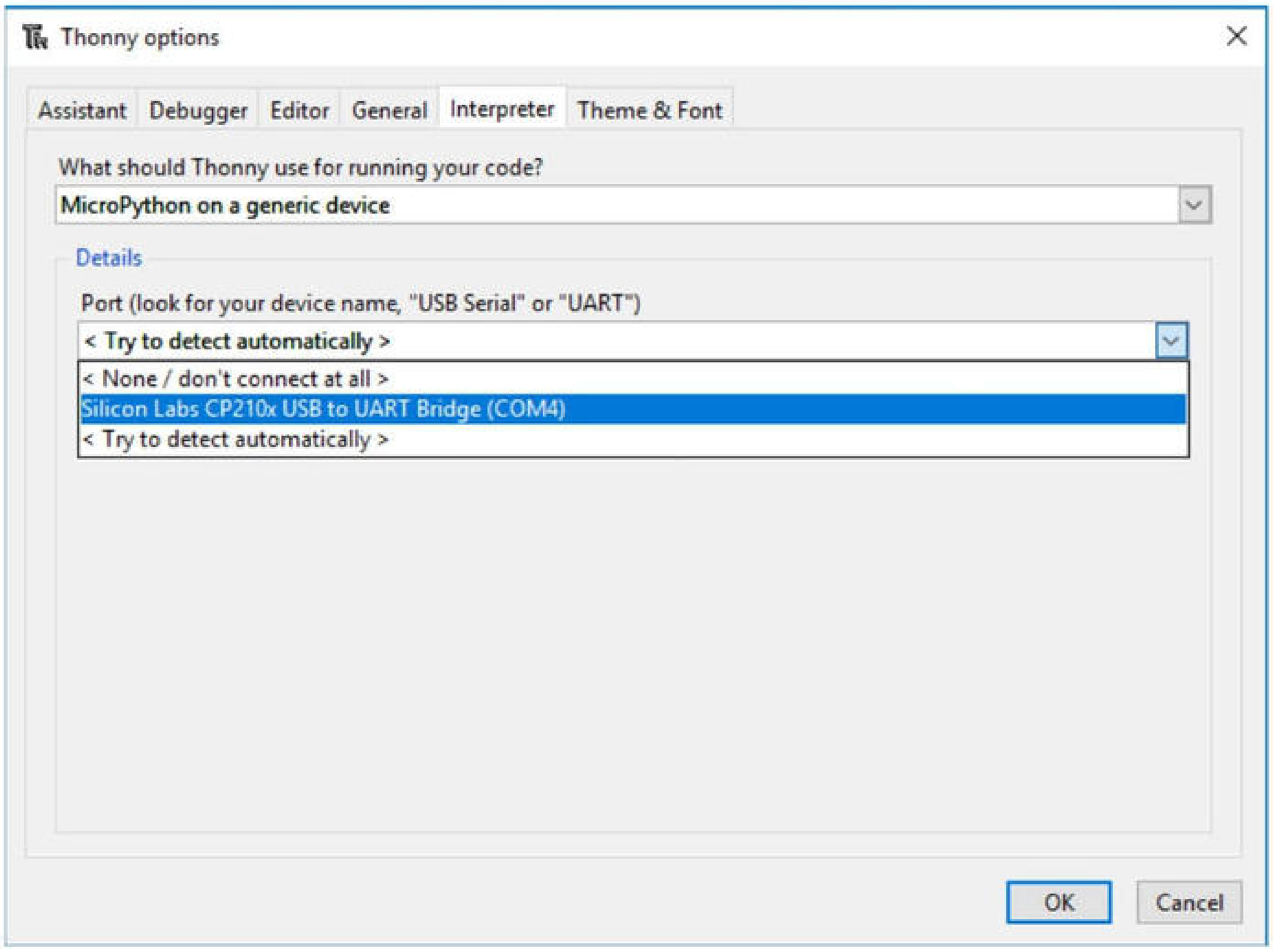
IMPORTANT: before testing the installation your ESP32/ESP8266 board needs to be flashed with MicroPython firmware.

Connect the board to your computer using an USB cable. To test the installation, you need to tell Thonny that you want to run MicroPython Interpreter and select the board you are using.

1. Go to **Tools** ▸ **Options** and select the **Interpreter** tab.
2. Choose **MicroPython on a generic device**.

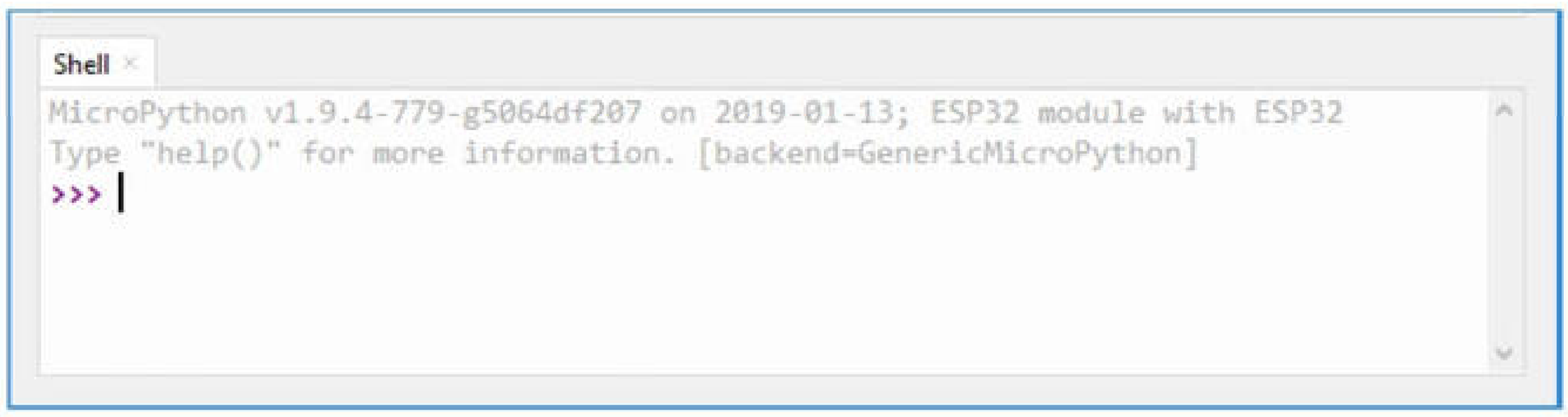


3. Then, select your device serial port (recommended).

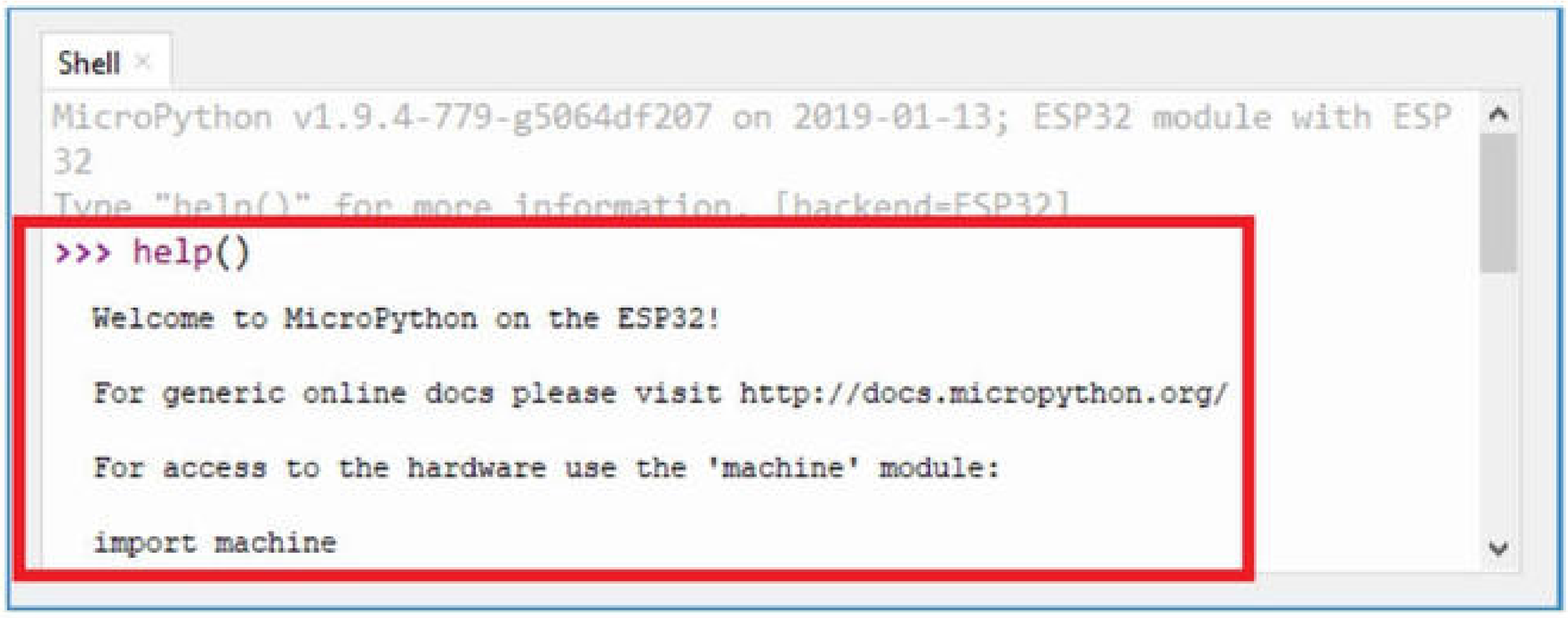


Note: you can also select the "**Try to detect automatically**" option, but only if you just have one board connected to your computer at a time. Otherwise, select the specific port for the board you're using.

4. Thonny IDE should now be connected to your board and you should see the prompt on the Shell.



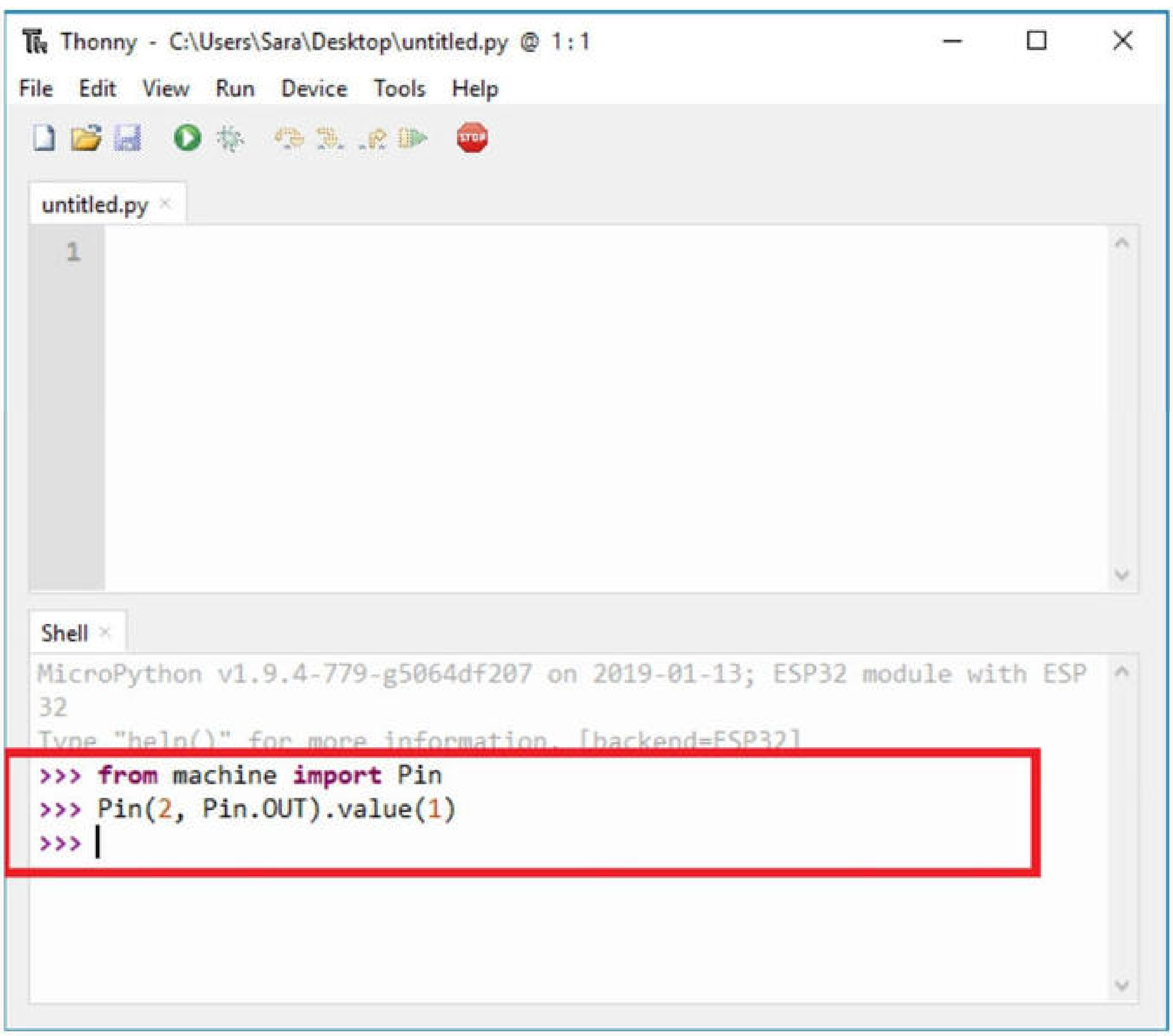
5. Type the command `help()` in the Shell and see if it responds back.



If it responded back, everything is working fine. Now, you can send a few more commands to test.

Send the following commands to light up the on-board LED:

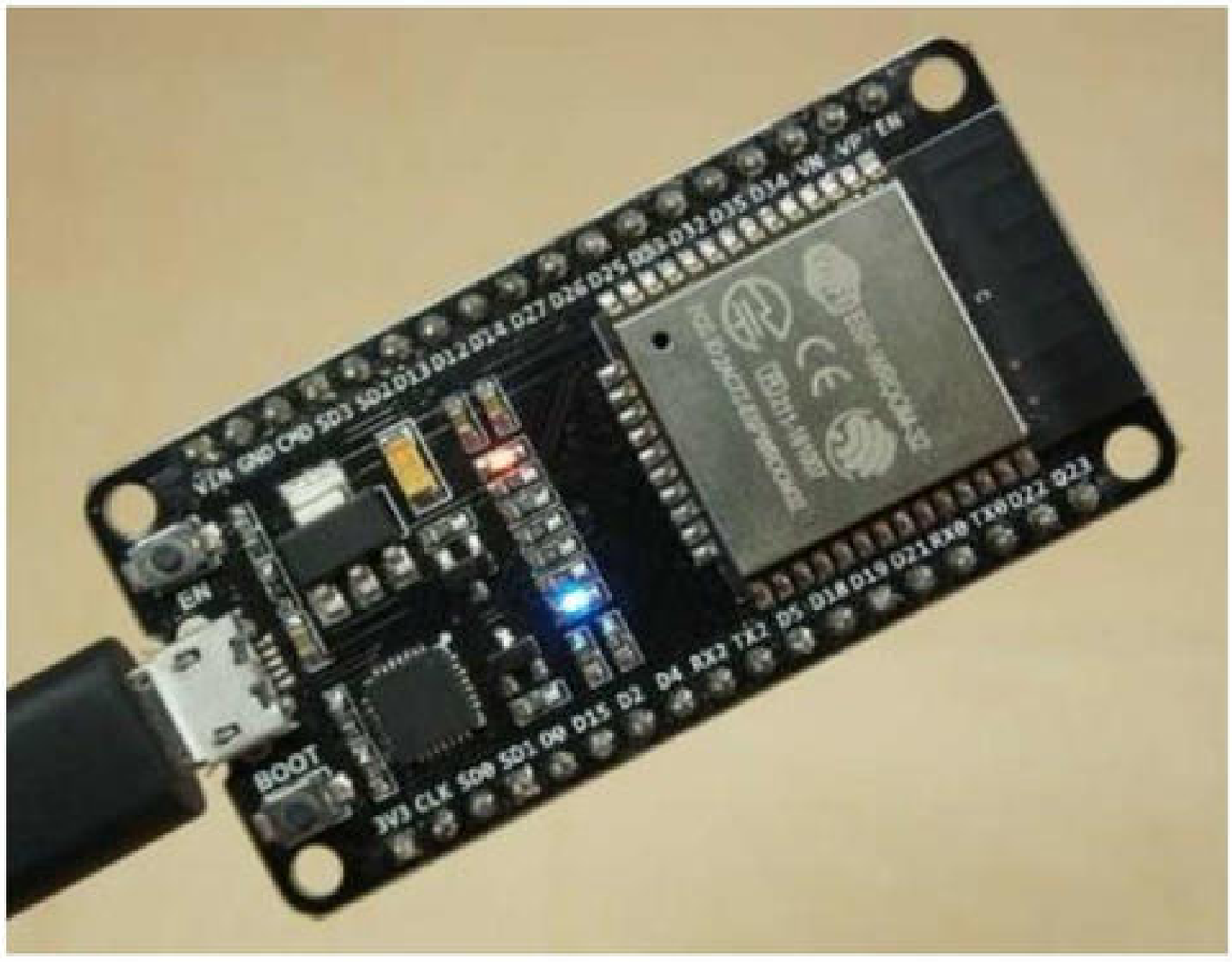
```
>>> from machine import Pin
>>> Pin(2, Pin.OUT).value(1)
```



Note: if you're using an ESP8266, the logic to turn on the LED works the other way around, so you should send the following command instead:

```
>>> Pin(2, Pin.OUT).value(0)
```

The on-board LED should light up.



Then, turn off the led:

```
>>> Pin(2, Pin.OUT).value(0)
```

Congratulations! Your installation was successful!

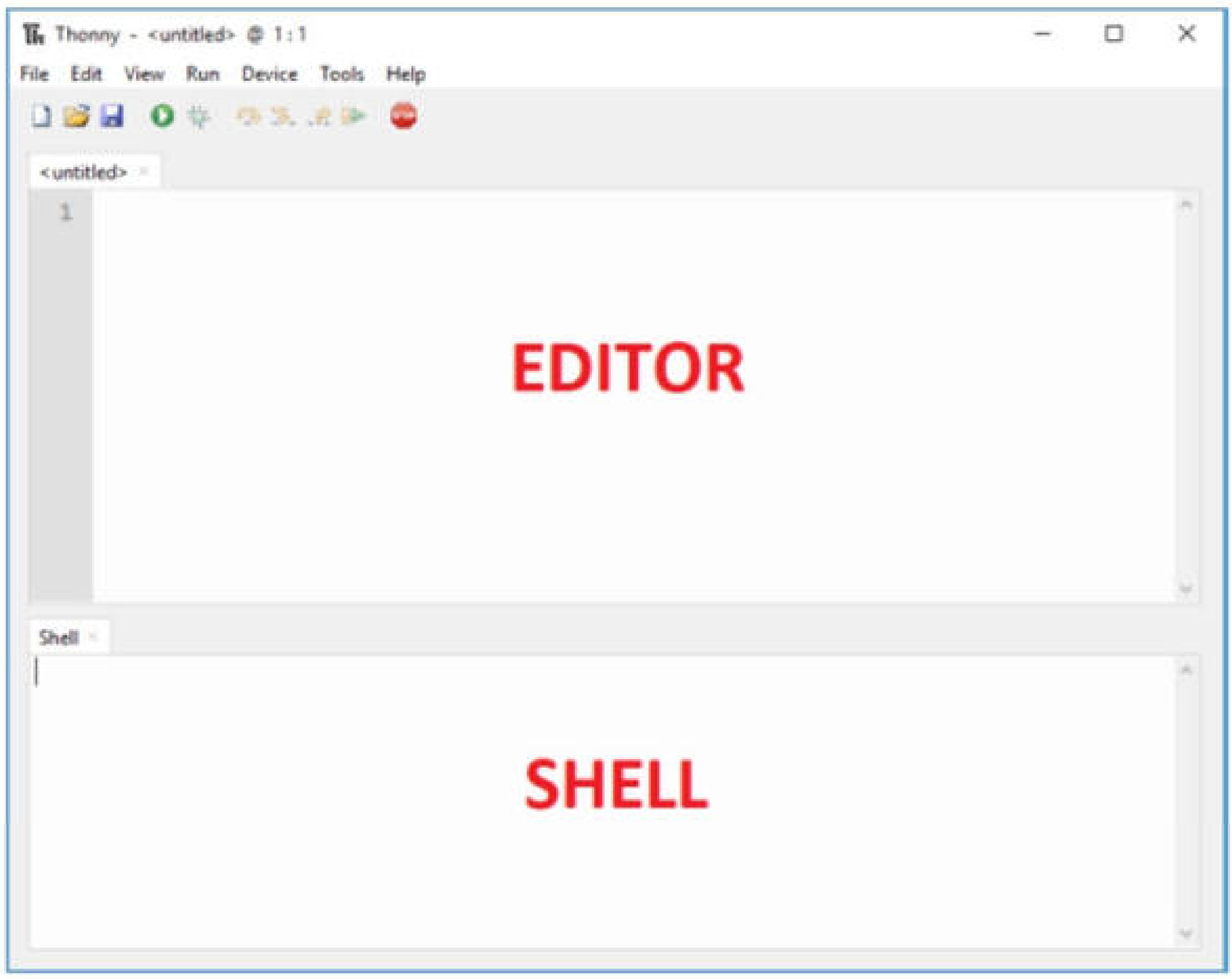
Thonny IDE Overview

This section is a quick overview on how to use Thonny IDE functionalities to program the ESP32 and ESP8266.

At this point, you should have:

- ESP32/ESP8266 flashed with MicroPython firmware
- Thonny IDE installed on your computer

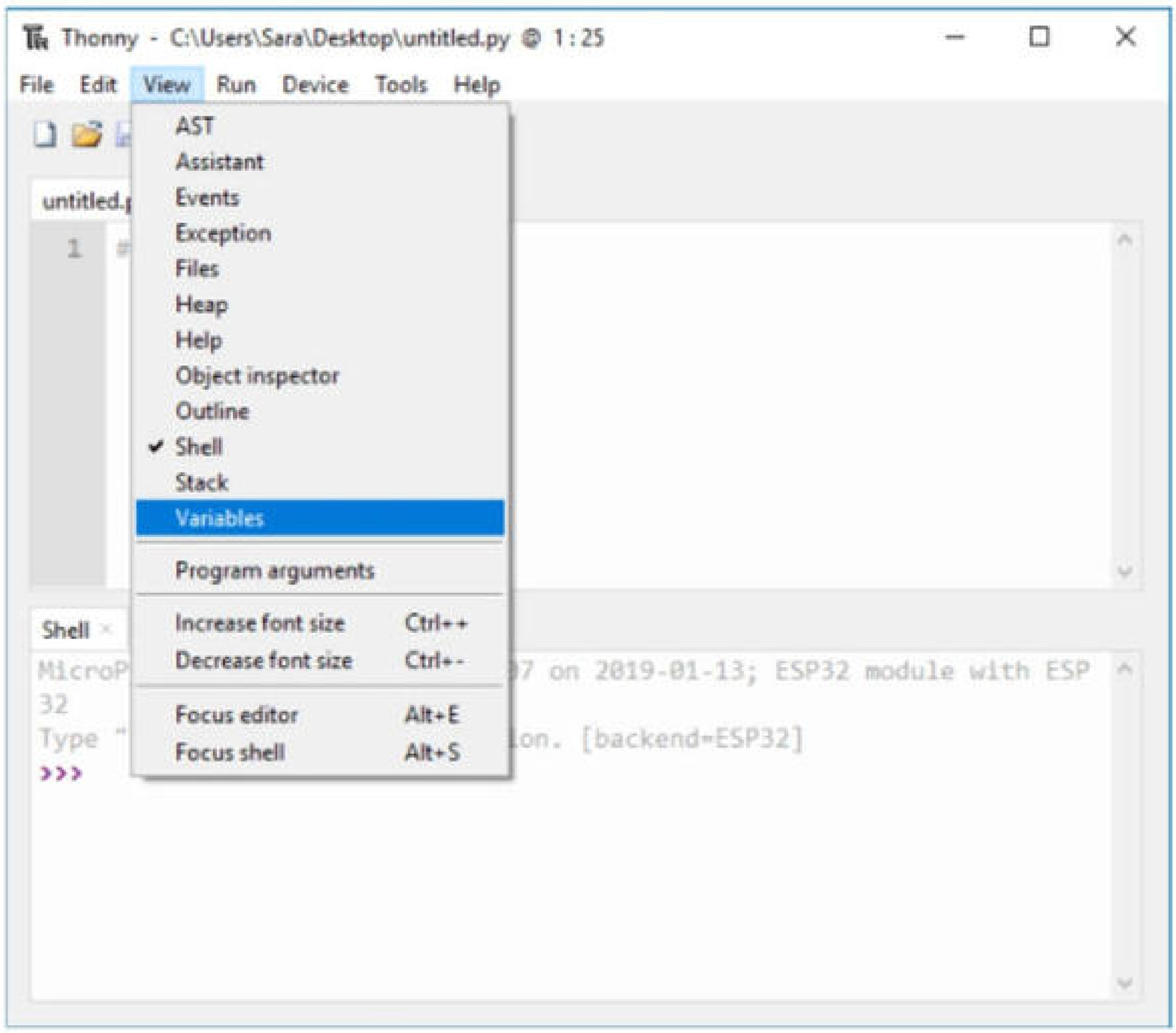
Open Thonny IDE. There are two different sections: Editor and MicroPython Shell/Terminal:



The Editor section is where you write your code and edit your *.py* files. You can open more than one file, and the Editor will open a new tab for each file.

In the MicroPython Shell you can type commands to be executed immediately by your ESP board without the need to upload new files. The terminal also provides information about the state of an executing program, shows errors related with upload, syntax errors, prints messages, etc...

You can also customize Thonny IDE to show other useful tabs. Go to **View** and you can select several tabs that provide more information.



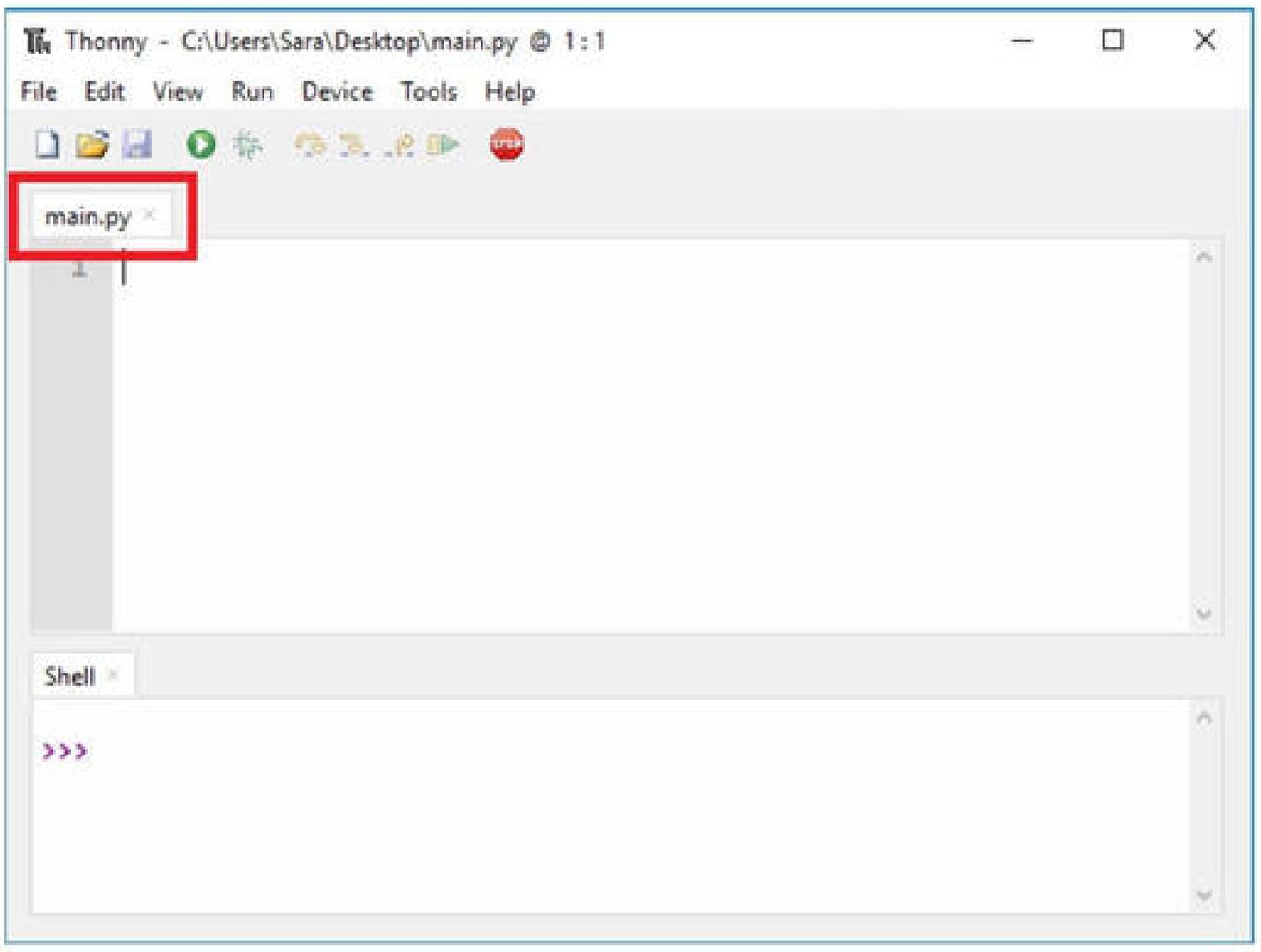
Something that could be very useful is the **Variables** tab. It lists all available variables in your program and their corresponding values.

Running Your First Script

To get you familiar with the process of writing a file and executing code on your ESP32/ESP8266 boards, we'll upload a new script that simply blinks the on-board LED of your ESP32 or ESP8266.

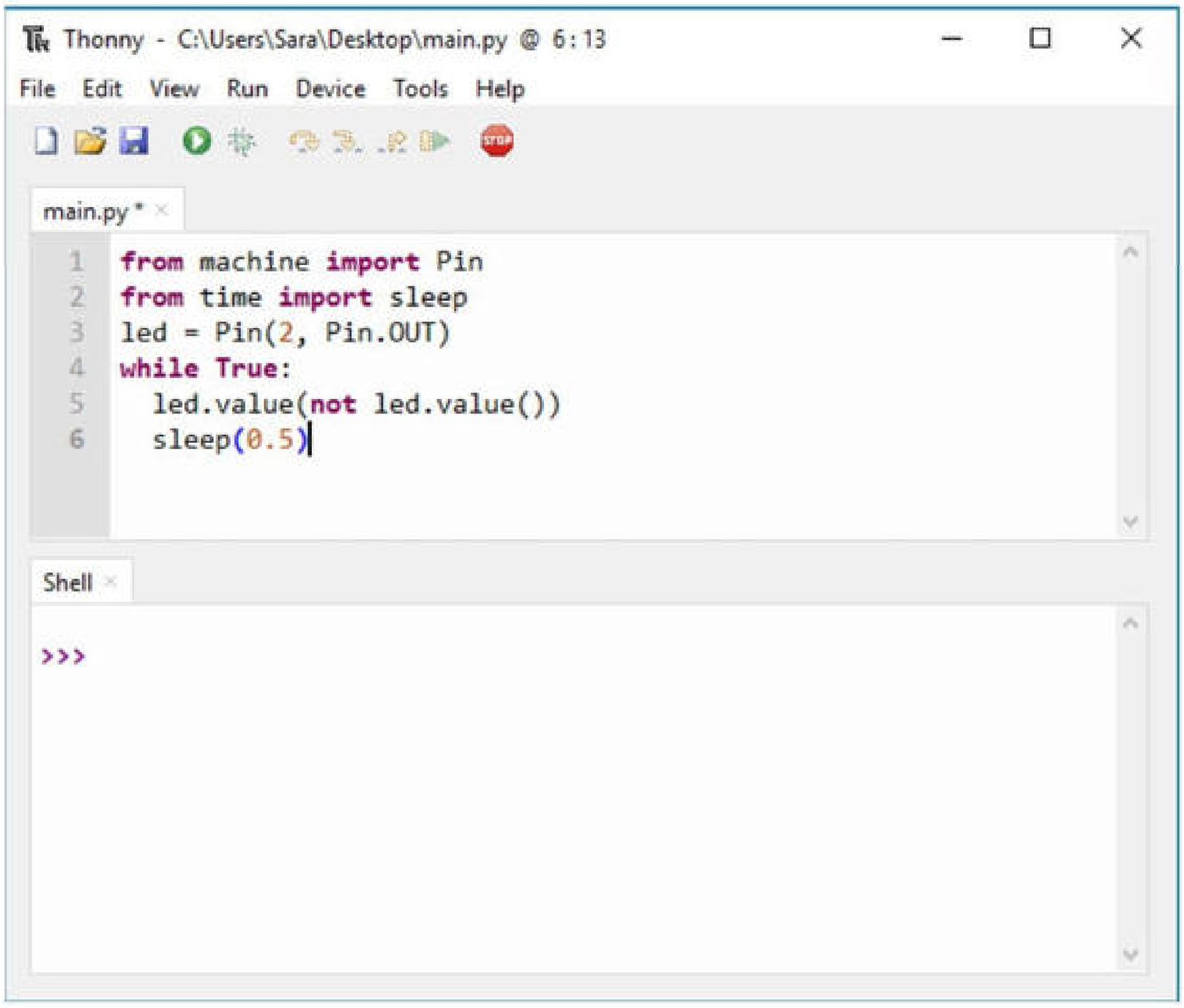
Creating the main.py file on your board

1. When you open Thonny IDE for the first time, the Editor shows an untitled file. Save that file as *main.py*. Simply, click the save icon and name the file *main.py*.
2. The Editor should now have a tab called *main.py*.



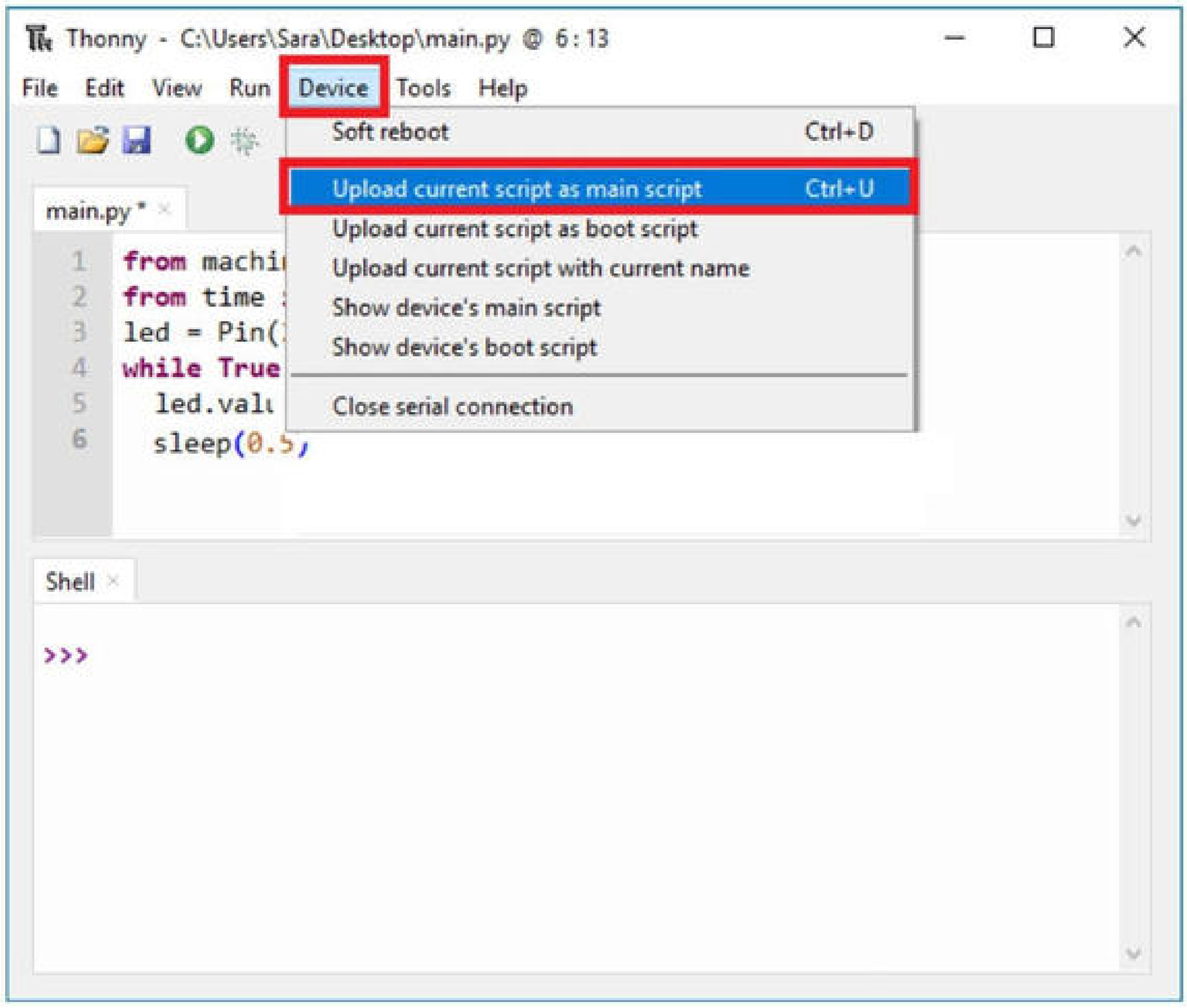
3. Copy the following code to the main.py file (make sure you copy with the right indentation):

```
from machine import Pin
from time import sleep
led = Pin(2, Pin.OUT)
while True:
  led.value(not led.value())
  sleep(0.5)
```

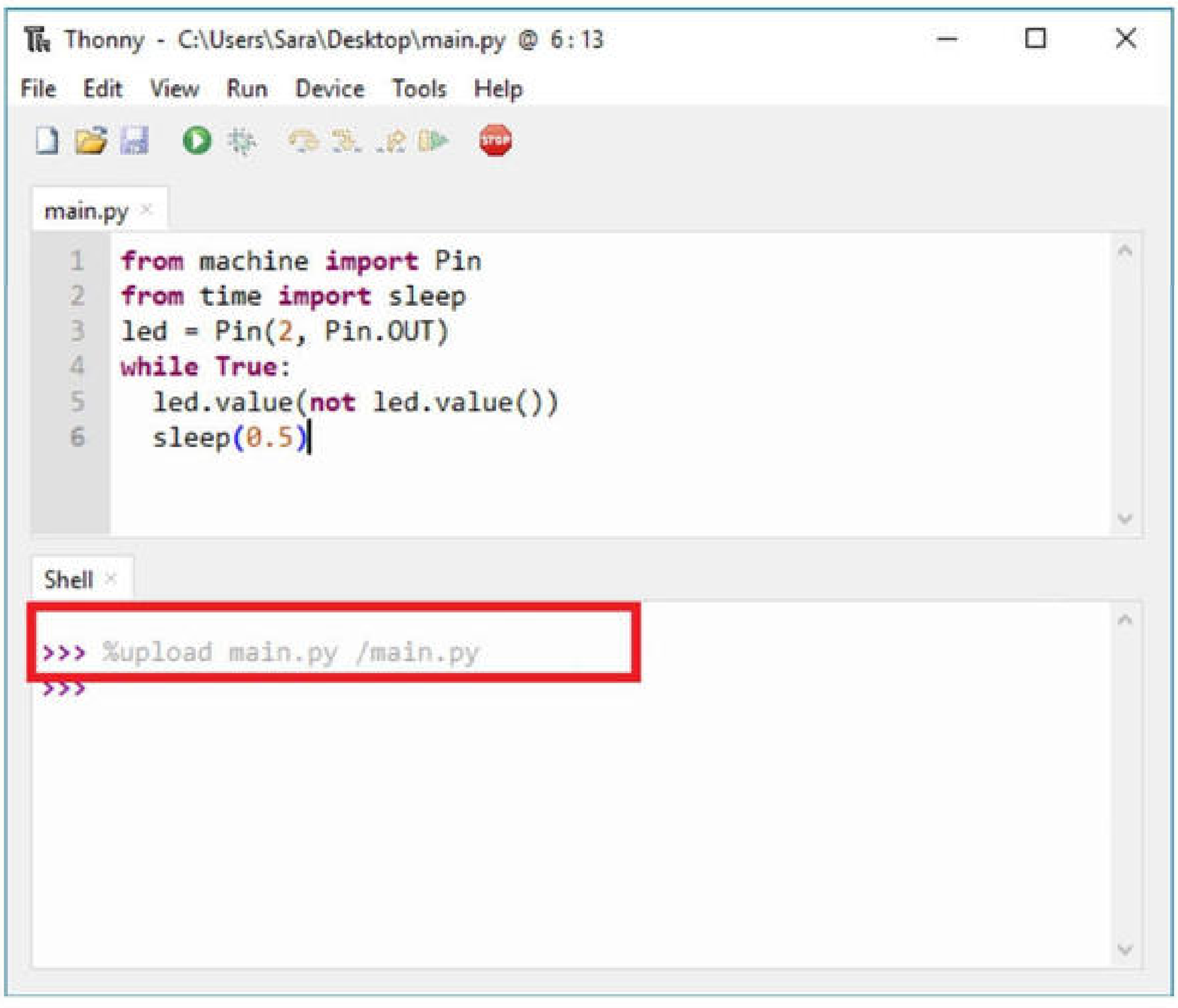


Uploading the Script

Go to **Device** and select **Upload current script as main script**.



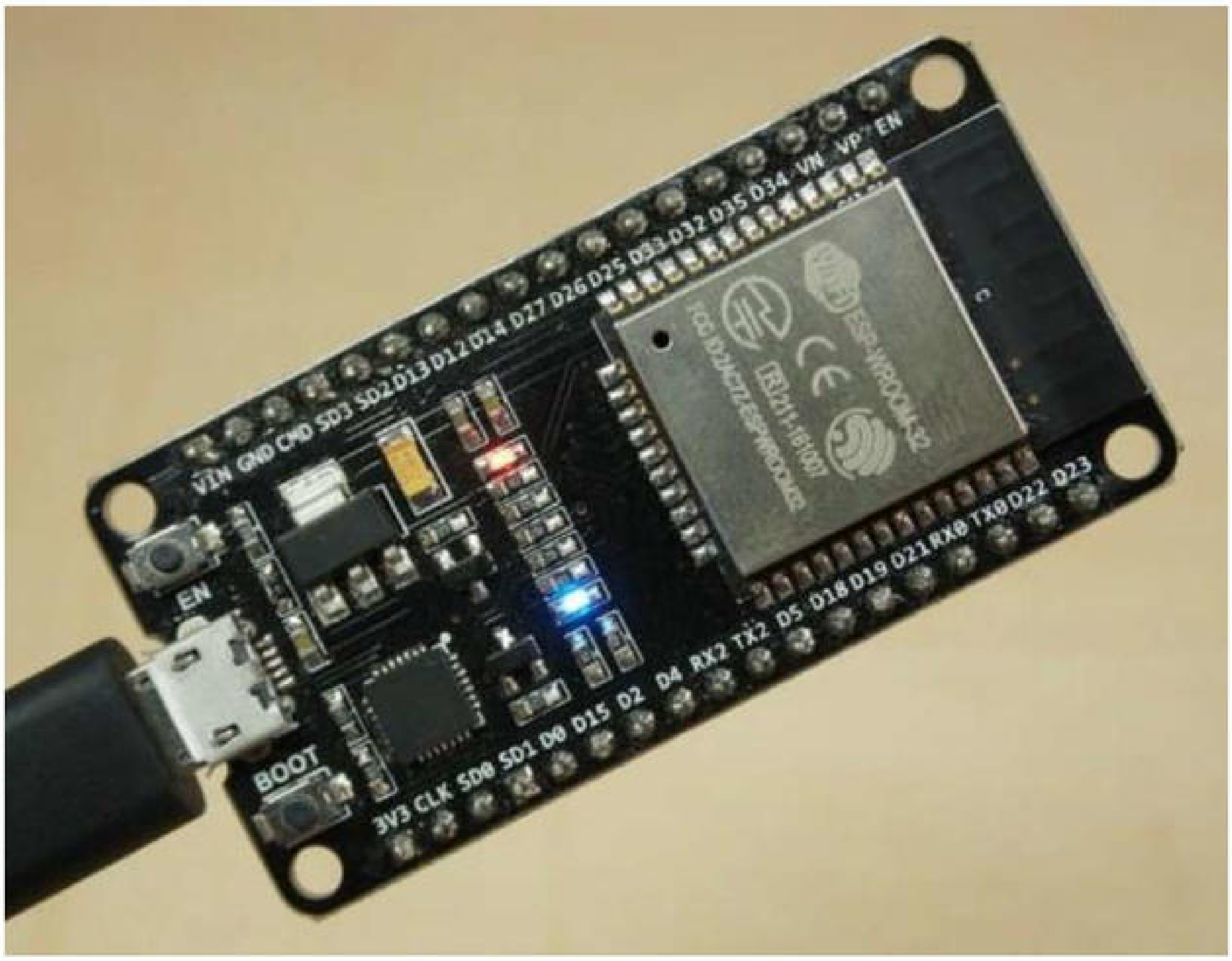
It should write the following on the Shell.



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

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

After uploading the script, press the ESP EN/RESET button. The ESP on-board LED should be blinking.

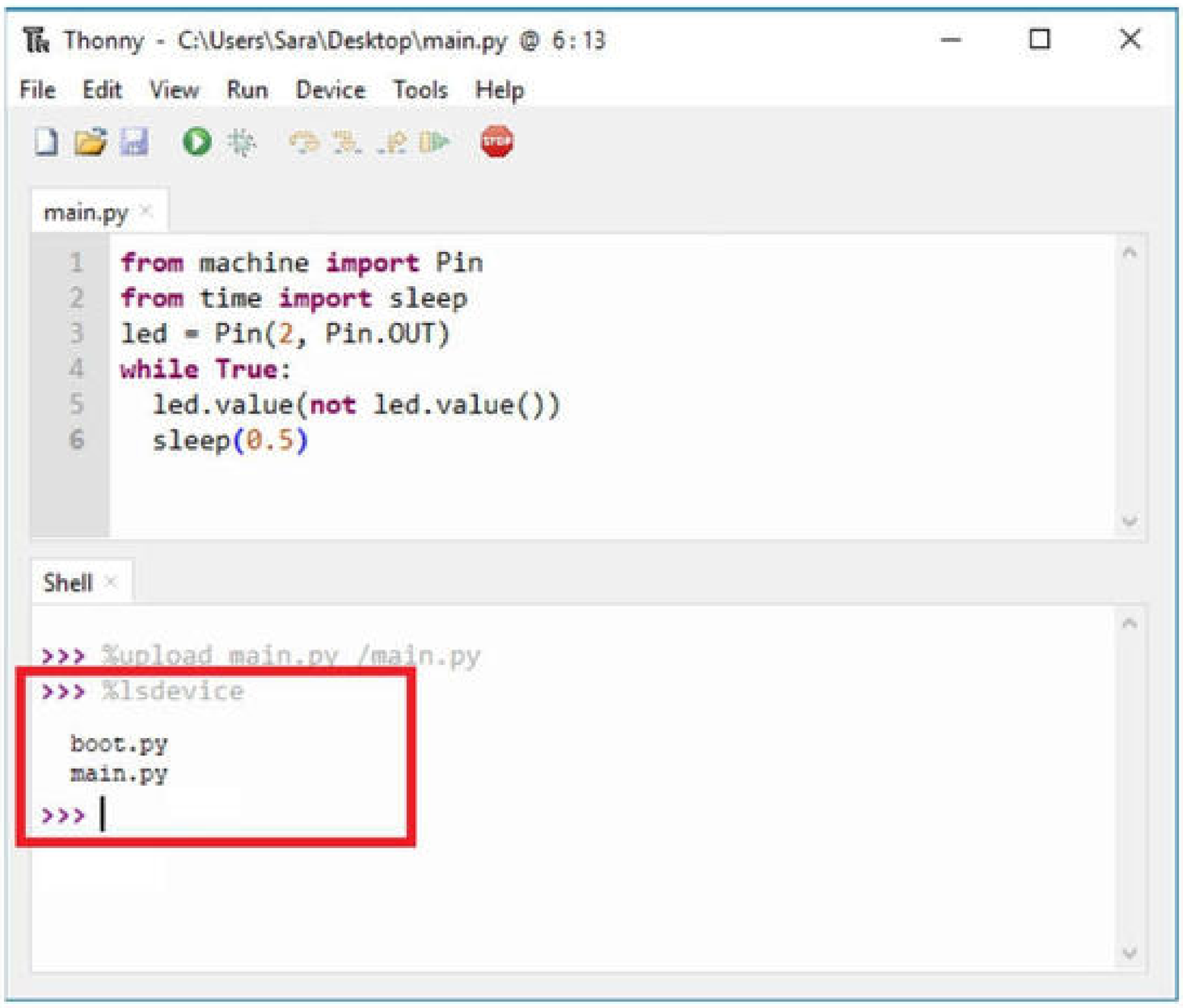


Congratulations, you've successfully uploaded a new script to your ESP32/ESP8266 using Thonny IDE.

List Files Saved on ESP32/ESP8266

To list all files saved on the ESP board, type in the Shell:

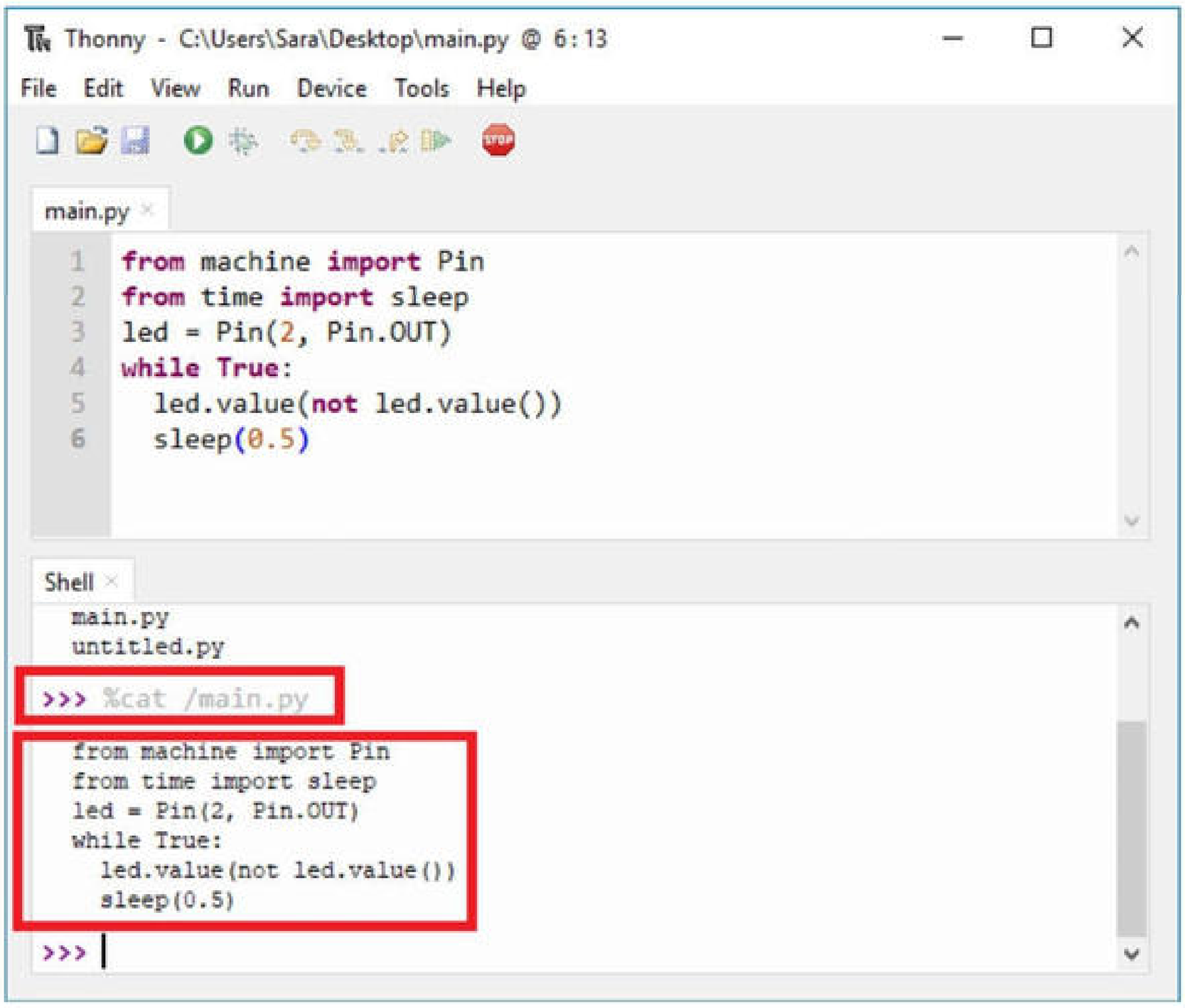
```
%lsdevice
```



Show File Content

To show the file content, use `%cat /` followed by the file path. For example:

```
%cat /main.py
```

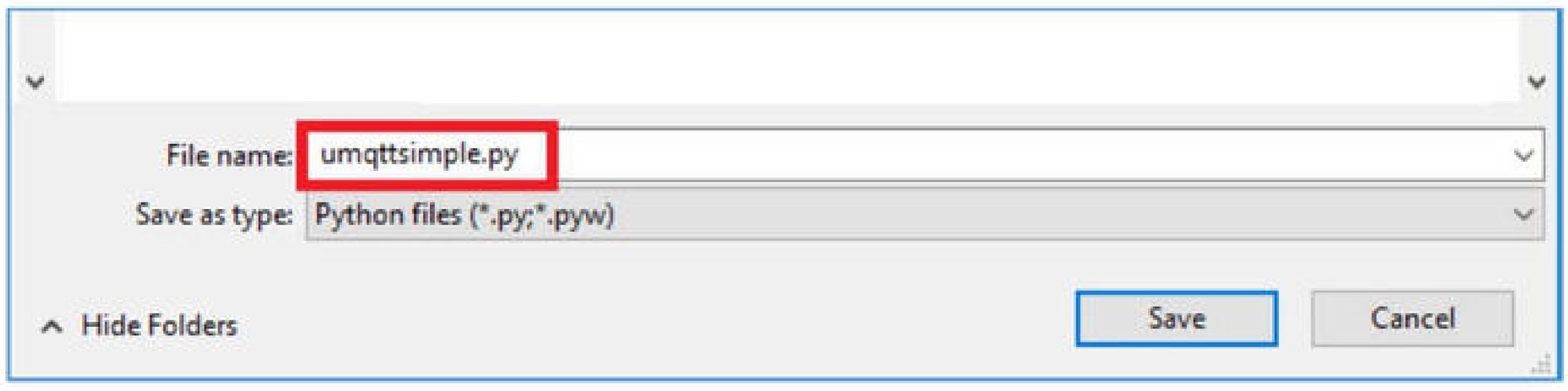


Alternatively, to show the *boot.py* and *main.py* scripts, you can also go to **Device** and select one of the options: **Show device's main script** or **Show device's boot script**.

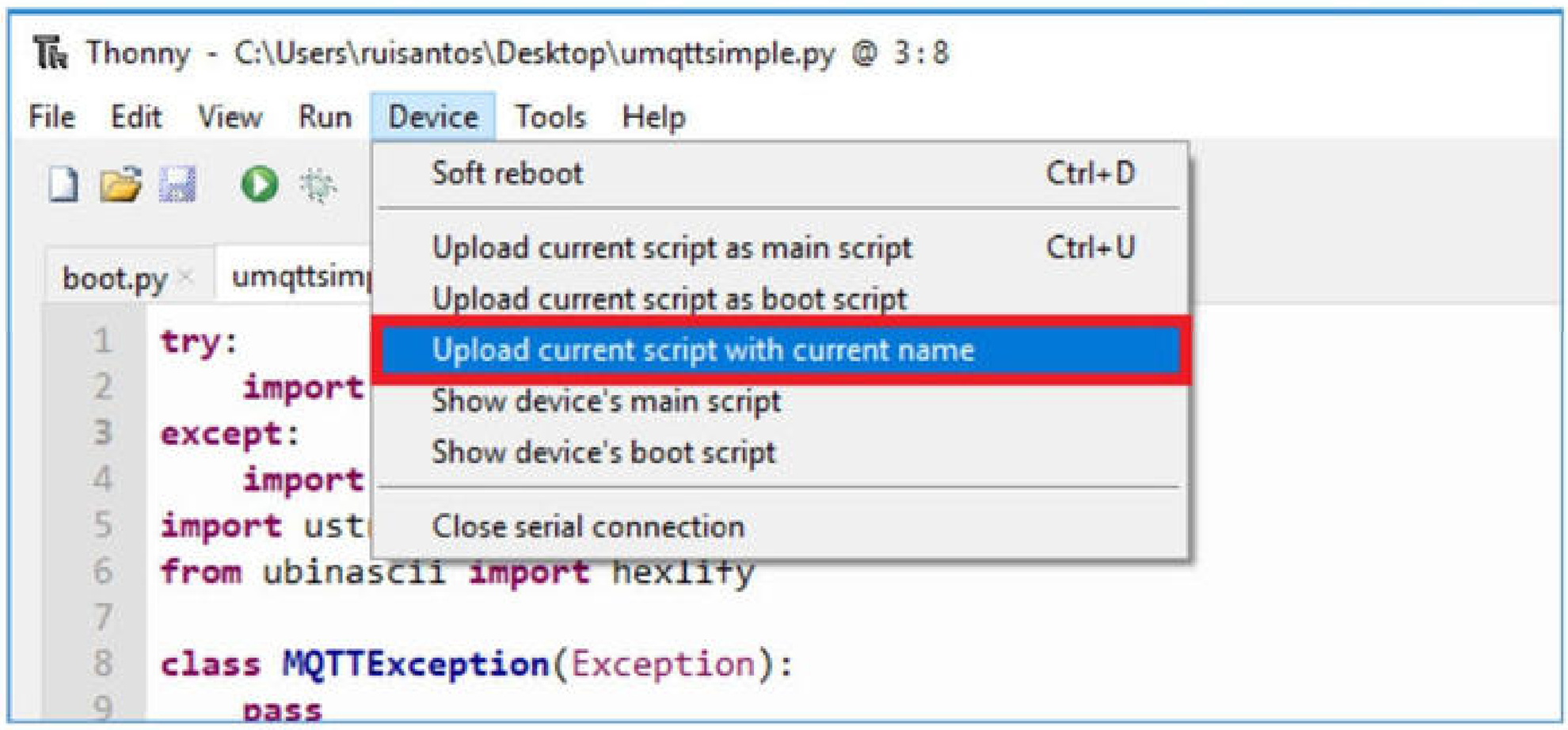
Uploading Files/Libraries

To upload a file with a unique name to your ESP using Thonny IDE, follow these next steps:

1. Create a new file;
2. Save it in your computer with the exact name that you want, for example "umqttsimple.py";



3. Go to **Device ▸ Upload current script with current name**;



The file should be saved on the ESP with the name "*umqttsimple.py*"

You can list the files stored in your ESP board with the command `%lsdevice`:

```
Shell
>>>
>>> %upload umqttsimple.py umqttsimple.py
>>> %lsdevice

  boot.py
  main.py
  umqttsimple.py

>>>
```

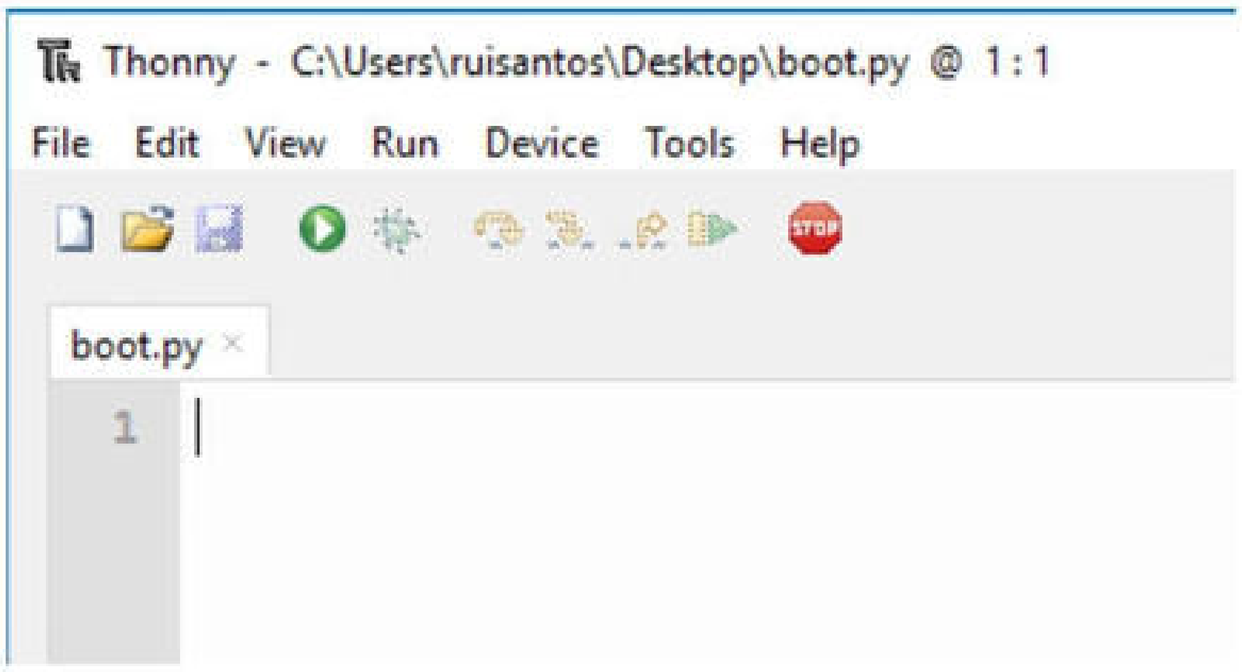
That's it! The *umqttsimple.py* file was uploaded to your board. You can see the file content using the `%cat /umqttsimple.py` command:

```
Shell
  boot.py
  main.py
  umqttsimple.py

>>> %cat /umqttsimple.py

  try:
      import usocket as socket
  except:
      import socket
  import ustruct as struct
  from ubinascii import hexlify

  class MQTTException(Exception):
      pass

  class MQTTClient:
```

Removing/Deleting File from ESP

At the moment, Thonny IDE doesn't have a command to remove a file from the ESP. So, in order to remove/delete all files completely from your ESP, you need to re-flash it with MicroPython firmware.

Note: you can upload a blank script to the ESP board to remove/delete code.



Troubleshooting Tips for Thonny IDE

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

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

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


Check the configuration, select Run → Stop/Restart or press
Ctrl+F2 to try again. (On some occasions it helps to wait before
trying again.)
```

Or:

```
========================== RESTART ==========================
Could not connect to REPL.
Make sure your device has suitable firmware and is not in
bootloader mode!
Disconnecting.
```

Or:

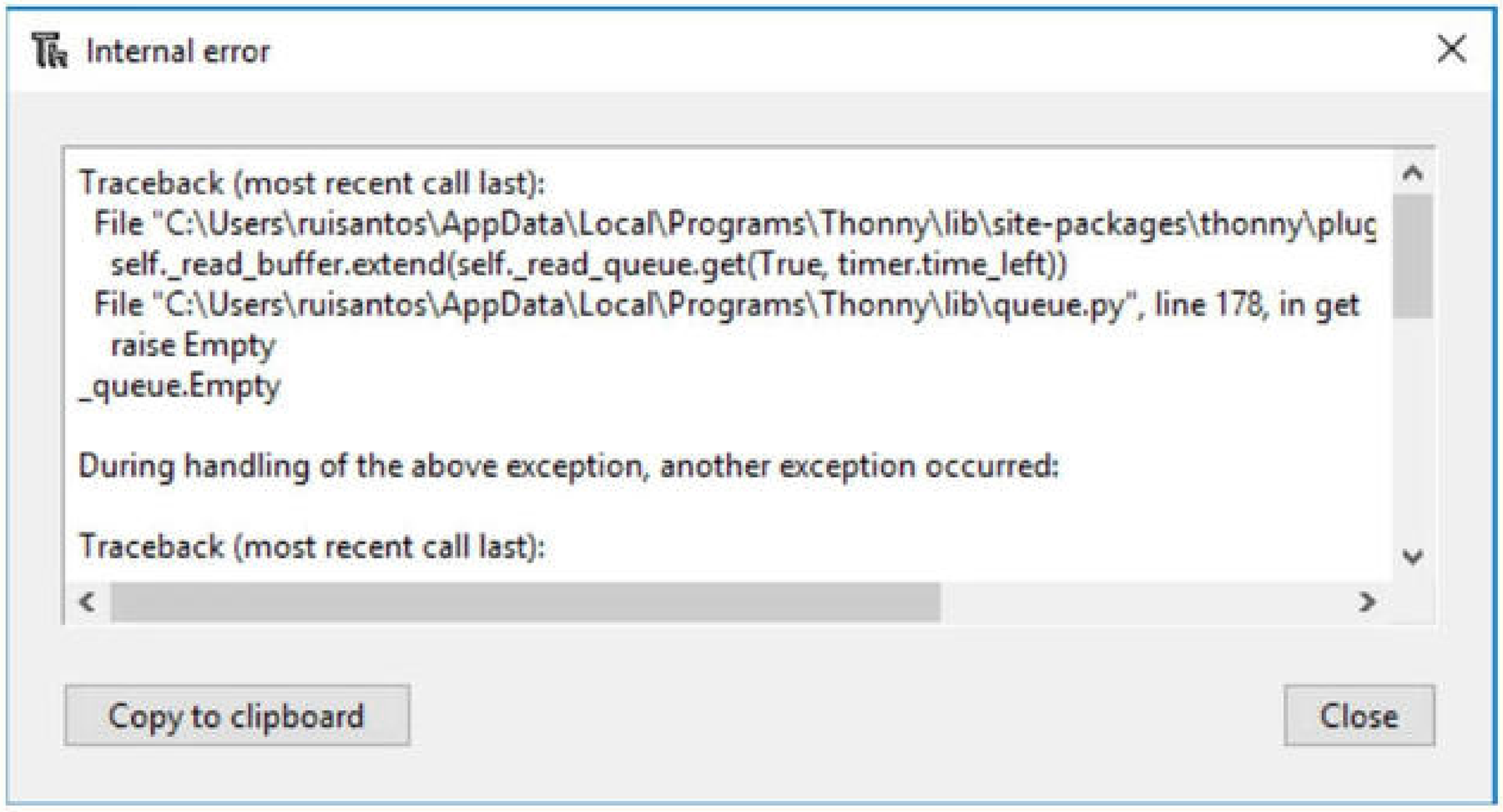
```
========================== RESTART ==========================
Lost connection to the device (EOF).
```

Unplug, and plug back your ESP board. Then, double-check that you've selected the right serial port in the **Tools ▸ Options ▸ Interpreter ▸ Port**. Then, click the "**Stop/Restart backend**" button to establish a serial communication. You should now be able to upload a new script or re-run new code.

These errors might also mean that you have your serial port being used in another program (like a serial terminal or in the Arduino IDE). Double-check that you've

closed all the programs that might be establishing a serial communication with your ESP board. Then, unplug and plug back your ESP board. Finally, restart Thonny IDE.

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



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

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

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

Error #4: *Problem restarting your ESP board, running a new script or opening the serial port*:

```
Brownout detector was triggered
```

Or if the ESP keeps restarting and printing the ESP boot information:

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

rst:0x1 (POWERON_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
```

```
configsip: 0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp
_drv:0x00
mode:DIO, clock div:2
load:0x3fff0018,len:4
load:0x3fff001c,len:4732
load:0x40078000,len:7496
load:0x40080400,len:5512
```

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

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

Solution: try a different shorter USB cable (with data wires), try a different computer USB port or use a USB hub with an external power supply.

Important: if you keep having constant problems or weird error messages, we recommend re-flashing your ESP board with the latest version of MicroPython firmware.

Error #5: *When I try to establish a serial communication with the ESP32/ESP8266 in Thonny IDE, I can't get it to connect.*

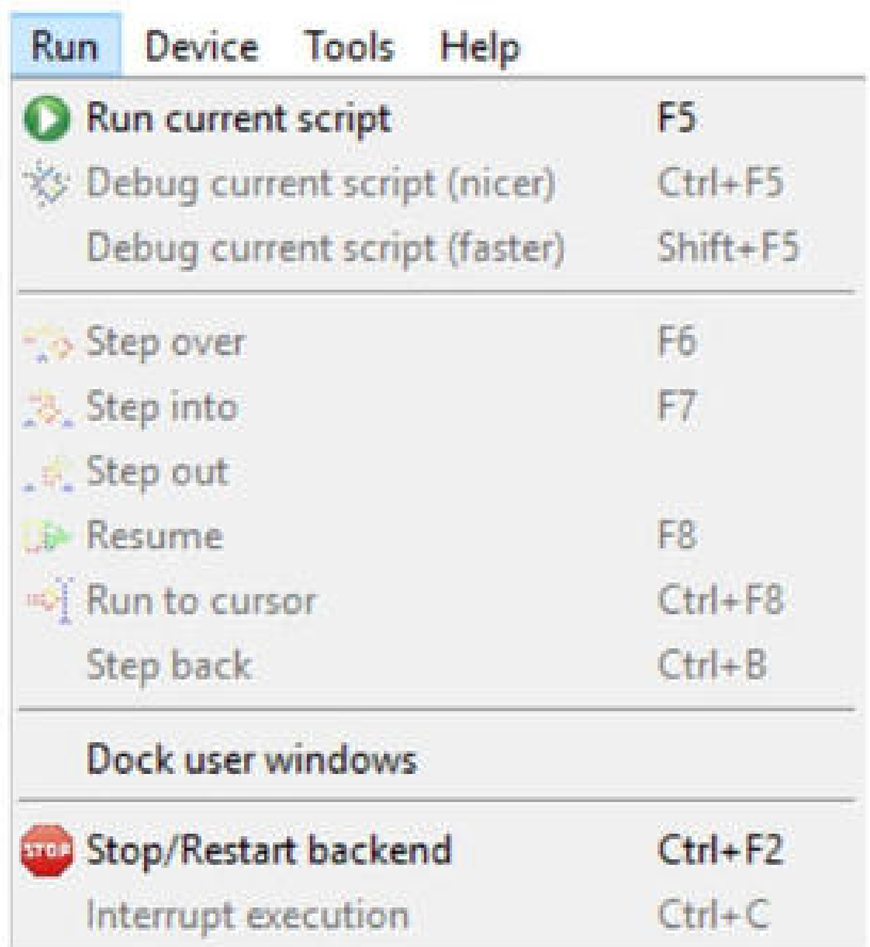
We think this is what's happening: when you're running a script in your board, sometimes it's busy running that script and performing the tasks.

So, you need to try start the connection by pressing "**Stop/Restart backend**" button multiple times or restart the ESP to catch available to establish the serial communication.

Warning: don't press the "**Stop/Restart backend**" button multiple times very quickly. After pressing that button, you need to be patient and wait a few seconds for the command to run.

If you're running a script that uses Wi-Fi, deep sleep, or it's doing multiple tasks, I recommend trying 3 or 4 times to establish the communication. If you can't, I recommend re-flash the ESP with MicroPython firmware.

Error #6: *Debug tools are grayed out*:



Thonny IDE debug tools aren't available for MicroPython. The debug tools are only available for the Python Interpreter, so being grayed out is the expected behavior.

Error #7: *You get one of the following indentation error messages:*

```
IndentationError: unexpected indent
```

Or

```
IndentationError: unindent doesn't match any outer indent level
```

Or

```
SyntaxError: invalid syntax
```

If you upload a new script to your ESP board and it prompts one of the preceding error messages, it means that there's a space or multiple spaces missing in your script. Python doesn't use *;* or *{}* to know where each line ends. Python uses indentation (everything is done using spaces). Double-check that all the spaces are properly inserted in your script. To avoid indentation errors, we recommend copying/pasting the code directly from the GitHub link provided in each example.

Final Thoughts

Congratulations for completing this course!

If you followed all the Modules presented in this course, you should be able to build your own projects by programming the ESP32/ESP8266 using MicroPython firmware. Besides learning MicroPython/Python programming basics, you also know how to:

- Interact with the ESP32/ESP8266 GPIOs: control outputs, read digital and analog inputs, produce PWM signals, set up interrupts, and more.
- Create a Web Server with the ESP32/ESP8266 to control outputs and display sensor readings.
- Use MQTT communication protocol to interact between two ESP boards.
- Build a user interface using Node-RED to control the ESP32/ESP8266 outputs and display sensor readings.
- Communicate with Node-RED using MQTT.
- Use different sensors like: PIR motion sensor, DHT11/22 temperature and humidity sensor, BME280 sensor module, and DS18B20 temperature sensor.
- Control other components like an OLED display, DC motor and addressable RGB LED strip.

Note: we didn't cover some interesting topics like other deep sleep methods and wake up sources, and bluetooth with the ESP32 because they are not fully supported at the moment. We intend to update this course with these subjects when we're sure they are stable and working at 100%.

We hope you had fun following this course and you've learned something new! If you have something that you would like to share (your projects, suggestions, feedback) let us know in the Private Forum or Facebook group. Good luck with all your projects,

Rui Santos and *Sara Santos*

Appendix: List of Parts Required

This appendix provides a list of all the electronics components used throughout this eBook/course.

Important: all components and parts are linked to our website MakerAdvisor.com/tools where you can compare prices in more than 8 different stores, so you can always find the best price.

List of components and parts:

- 2x ESP32 or 2x ESP8266 boards
    - Best ESP32 Development Board
    - Best ESP8266 Development Board
- Breadboard
- Jumper wires
- LEDs
- Resistors: 330 Ω, 10k Ω, 4.7k Ω
- Pushbutton
- Potentiometer
- PIR motion sensor (AM312) or PIR motion sensor (HC-SR501)
- DHT11/DHT22 temperature and humidity sensor
- BME280 temperature, humidity, and pressure sensor module
- DS18B20 temperature sensor
- L298N motor driver
- Mini DC Motor
- 0.1 µF ceramic capacitor
- SSD1306 128x64 - 0.96 inch OLED display
- WS2812B addressable RGB LED strip
- Logic level converter module

Additional parts required for MQTT Units:

- Raspberry Pi - Best Raspberry Pi Starter Kits
- MicroSD card
- Raspberry Pi power supply

Other RNT Courses/eBooks

Random Nerd Tutorials is an online resource with electronics projects, tutorials and reviews. Currently, Random Nerd Tutorials has more than 200 free blog posts with complete tutorials using open-source hardware that anyone can read, remix and apply to their own projects.

To keep free tutorials coming, there's also paid content or as we like to call "premium content". To support Random Nerd Tutorials, you can download premium content here. If you enjoy this eBook, make sure you check all our courses and resources.

Here's a list with all the courses/eBooks available to download at Random Nerd Tutorials:

- Learn ESP32 with Arduino IDE (Bestseller)
- 20 Easy Raspberry Pi Projects (available on Amazon)
- Build a Home Automation System for $100
- Home Automation Using ESP8266 (3rd Edition)
- Arduino Step-by-step Projects Course
- Android Apps for Arduino with MIT App Inventor 2
- Electronics For Beginners eBook