# MicroPython

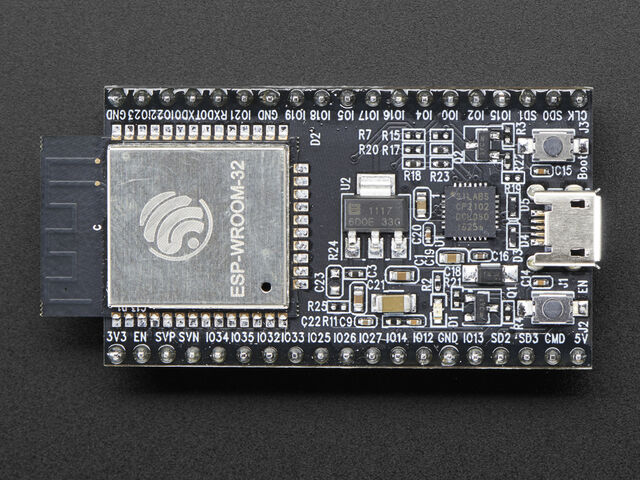
[**http://micropython.org/**](http://micropython.org/)

MicroPython is a lean and efficient implementation of the [Python 3](http://www.python.org/) programming language that includes a small subset of the Python standard library and is optimised to run on microcontrollers and in constrained environments.

The MicroPython [pyboard](http://micropython.org/pyboard) is a compact electronic circuit board that runs MicroPython on the bare metal, giving you a low-level Python operating system that can be used to control all kinds of electronic projects.

MicroPython is packed full of advanced features such as an interactive prompt, arbitrary precision integers, closures, list comprehension, generators, exception handling and more. Yet it is compact enough to fit and run within just 256k of code space and 16k of RAM.

MicroPython aims to be as compatible with normal Python as possible to allow you to transfer code with ease from the desktop to a microcontroller or embedded system.

[](https://docs.micropython.org/en/latest/_images/esp32.jpg)

The Espressif ESP32 Development Board (image attribution: Adafruit).

**목차**

[MicroPython 1](#_Toc31567069)

[MicroPython license information 8](#_Toc31567070)

[Quick reference for the ESP32 12](#_Toc31567071)

[Installing MicroPython 12](#_Toc31567072)

[General board control 12](#_Toc31567073)

[Networking 13](#_Toc31567074)

[Delay and timing 14](#_Toc31567075)

[Timers 14](#_Toc31567076)

[Pins and GPIO 14](#_Toc31567077)

[PWM (pulse width modulation) 15](#_Toc31567078)

[ADC (analog to digital conversion) 15](#_Toc31567079)

[Software SPI bus 16](#_Toc31567080)

[Hardware SPI bus 17](#_Toc31567081)

[I2C bus 17](#_Toc31567082)

[Real time clock (RTC) 18](#_Toc31567083)

[Deep-sleep mode 18](#_Toc31567084)

[RMT 18](#_Toc31567085)

[OneWire driver 19](#_Toc31567086)

[NeoPixel driver 19](#_Toc31567087)

[Capacitive touch 19](#_Toc31567088)

[DHT driver 20](#_Toc31567089)

[WebREPL (web browser interactive prompt) 20](#_Toc31567090)

[MicroPython language and implementation 22](#_Toc31567091)

[Glossary 23](#_Toc31567092)

[The MicroPython Interactive Interpreter Mode (aka REPL) 25](#_Toc31567093)

[Auto-indent 25](#_Toc31567094)

[Auto-completion 25](#_Toc31567095)

[Interrupting a running program 26](#_Toc31567096)

[Paste mode 26](#_Toc31567097)

[Soft reset 26](#_Toc31567098)

[The special variable \_ (underscore) 27](#_Toc31567099)

[Raw mode 27](#_Toc31567100)

[MicroPython .mpy files 28](#_Toc31567101)

[Versioning and compatibility of .mpy files 28](#_Toc31567102)

[Binary encoding of .mpy files 30](#_Toc31567103)

[The header 30](#_Toc31567104)

[Raw code elements 30](#_Toc31567105)

[Writing interrupt handlers 32](#_Toc31567106)

[Tips and recommended practices 32](#_Toc31567107)

[MicroPython issues 32](#_Toc31567108)

[The emergency exception buffer 32](#_Toc31567109)

[Simplicity 32](#_Toc31567110)

[Communication between an ISR and the main program 32](#_Toc31567111)

[The use of object methods as callbacks 33](#_Toc31567112)

[Creation of Python objects 33](#_Toc31567113)

[Use of Python objects 34](#_Toc31567114)

[Overcoming the float limitation 34](#_Toc31567115)

[Using micropython.schedule 34](#_Toc31567116)

[Exceptions 35](#_Toc31567117)

[General issues 35](#_Toc31567118)

[Interrupt handler design 35](#_Toc31567119)

[Reentrancy 36](#_Toc31567120)

[Critical sections 36](#_Toc31567121)

[Interrupts and the REPL 37](#_Toc31567122)

[Maximising MicroPython speed 39](#_Toc31567123)

[Designing for speed 39](#_Toc31567124)

[Algorithms 39](#_Toc31567125)

[RAM allocation 40](#_Toc31567126)

[Buffers 40](#_Toc31567127)

[Floating point 40](#_Toc31567128)

[Arrays 40](#_Toc31567129)

[Identifying the slowest section of code 41](#_Toc31567130)

[MicroPython code improvements 41](#_Toc31567131)

[The const() declaration 41](#_Toc31567132)

[Caching object references 41](#_Toc31567133)

[Controlling garbage collection 41](#_Toc31567134)

[The Native code emitter 41](#_Toc31567135)

[The Viper code emitter 42](#_Toc31567136)

[Accessing hardware directly 43](#_Toc31567137)

[MicroPython on microcontrollers 45](#_Toc31567138)

[Flash memory 45](#_Toc31567139)

[RAM 45](#_Toc31567140)

[Compilation phase 45](#_Toc31567141)

[Execution phase 46](#_Toc31567142)

[The heap 48](#_Toc31567143)

[Fragmentation 49](#_Toc31567144)

[Reporting 49](#_Toc31567145)

[Control of garbage collection 50](#_Toc31567146)

[String operations 50](#_Toc31567147)

[Postscript 51](#_Toc31567148)

[Footnote: gc.collect() return value 51](#_Toc31567149)

[Distribution packages, package management, and deploying applications 52](#_Toc31567150)

[Overview 52](#_Toc31567151)

[Distribution packages 52](#_Toc31567152)

[upip package manager 53](#_Toc31567153)

[Cross-installing packages 54](#_Toc31567154)

[Cross-installing packages with freezing 54](#_Toc31567155)

[Creating distribution packages 55](#_Toc31567156)

[Application resources 56](#_Toc31567157)

[References 57](#_Toc31567158)

[Inline assembler for Thumb2 architectures 58](#_Toc31567159)

[Document conventions 58](#_Toc31567160)

[Instruction categories 58](#_Toc31567161)

[Usage examples 59](#_Toc31567162)

[References 59](#_Toc31567163)

[Working with filesystems 60](#_Toc31567164)

[VFS 60](#_Toc31567165)

[Block devices 61](#_Toc31567166)

[Built-in block devices 61](#_Toc31567167)

[Custom block devices 61](#_Toc31567168)

[Filesystems 63](#_Toc31567169)

[FAT 63](#_Toc31567170)

[Littlefs 64](#_Toc31567171)

[Hybrid (STM32) 64](#_Toc31567172)

[Hybrid (ESP32) 65](#_Toc31567173)

[The pyboard.py tool 66](#_Toc31567174)

[Running a command on the device 66](#_Toc31567175)

[Running a script on the device 66](#_Toc31567176)

[Filesystem access 67](#_Toc31567177)

[Using the pyboard library 67](#_Toc31567178)

[MicroPython libraries 69](#_Toc31567179)

[Python standard libraries and micro-libraries 69](#_Toc31567180)

[MicroPython-specific libraries 70](#_Toc31567181)

[Port-specific libraries 70](#_Toc31567182)

[Libraries specific to the pyboard 71](#_Toc31567183)

[Libraries specific to the WiPy 71](#_Toc31567184)

[Libraries specific to the ESP8266 and ESP32 71](#_Toc31567185)

[MicroPython differences from CPython 73](#_Toc31567186)

[Syntax 74](#_Toc31567187)

[Spaces 74](#_Toc31567188)

[uPy requires spaces between literal numbers and keywords, CPy doesn’t 74](#_Toc31567189)

[Unicode 74](#_Toc31567190)

[Unicode name escapes are not implemented 74](#_Toc31567191)

[Core language 75](#_Toc31567192)

[Classes 75](#_Toc31567193)

[Special method \_\_del\_\_ not implemented for user-defined classes 75](#_Toc31567194)

[Method Resolution Order (MRO) is not compliant with CPython 75](#_Toc31567195)

[When inheriting from multiple classes super() only calls one class 75](#_Toc31567196)

[Calling super() getter property in subclass will return a property object, not the value 76](#_Toc31567197)

[Functions 76](#_Toc31567198)

[Error messages for methods may display unexpected argument counts 76](#_Toc31567199)

[User-defined attributes for functions are not supported 76](#_Toc31567200)

[Generator 77](#_Toc31567201)

[Context manager \_\_exit\_\_() not called in a generator which does not run to completion 77](#_Toc31567202)

[Runtime 77](#_Toc31567203)

[Local variables aren’t included in locals() result 77](#_Toc31567204)

[Code running in eval() function doesn’t have access to local variables 77](#_Toc31567205)

[import 78](#_Toc31567206)

[\_\_path\_\_ attribute of a package has a different type (single string instead of list of strings) in MicroPython 78](#_Toc31567207)

[Failed to load modules are still registered as loaded 78](#_Toc31567208)

[MicroPython does’t support namespace packages split across filesystem. 78](#_Toc31567209)

[Builtin types 80](#_Toc31567210)

[Exception 80](#_Toc31567211)

[Exception chaining not implemented 80](#_Toc31567212)

[User-defined attributes for builtin exceptions are not supported 80](#_Toc31567213)

[Exception in while loop condition may have unexpected line number 80](#_Toc31567214)

[Exception.\_\_init\_\_ method does not exist. 80](#_Toc31567215)

[bytearray 81](#_Toc31567216)

[Array slice assignment with unsupported RHS 81](#_Toc31567217)

[bytes 81](#_Toc31567218)

[bytes objects support .format() method 81](#_Toc31567219)

[bytes() with keywords not implemented 81](#_Toc31567220)

[Bytes subscription with step != 1 not implemented 82](#_Toc31567221)

[dict 82](#_Toc31567222)

[Dictionary keys view does not behave as a set. 82](#_Toc31567223)

[float 82](#_Toc31567224)

[uPy and CPython outputs formats may differ 82](#_Toc31567225)

[int 82](#_Toc31567226)

[No int conversion for int-derived types available 82](#_Toc31567227)

[list 83](#_Toc31567228)

[List delete with step != 1 not implemented 83](#_Toc31567229)

[List slice-store with non-iterable on RHS is not implemented 83](#_Toc31567230)

[List store with step != 1 not implemented 83](#_Toc31567231)

[str 83](#_Toc31567232)

[Start/end indices such as str.endswith(s, start) not implemented 83](#_Toc31567233)

[Attributes/subscr not implemented 84](#_Toc31567234)

[str(...) with keywords not implemented 84](#_Toc31567235)

[str.ljust() and str.rjust() not implemented 84](#_Toc31567236)

[None as first argument for rsplit such as str.rsplit(None, n) not implemented 84](#_Toc31567237)

[Instance of a subclass of str cannot be compared for equality with an instance of a str 84](#_Toc31567238)

[Subscript with step != 1 is not yet implemented 85](#_Toc31567239)

[tuple 85](#_Toc31567240)

[Tuple load with step != 1 not implemented 85](#_Toc31567241)

[Modules 86](#_Toc31567242)

[array 86](#_Toc31567243)

[Looking for integer not implemented 86](#_Toc31567244)

[Array deletion not implemented 86](#_Toc31567245)

[Subscript with step != 1 is not yet implemented 86](#_Toc31567246)

[builtins 86](#_Toc31567247)

[Second argument to next() is not implemented 86](#_Toc31567248)

[deque 87](#_Toc31567249)

[Deque not implemented 87](#_Toc31567250)

[json 87](#_Toc31567251)

[JSON module does not throw exception when object is not serialisable 87](#_Toc31567252)

[struct 87](#_Toc31567253)

[Struct pack with too few args, not checked by uPy 87](#_Toc31567254)

[Struct pack with too many args, not checked by uPy 88](#_Toc31567255)

[sys 88](#_Toc31567256)

[Overriding sys.stdin, sys.stdout and sys.stderr not possible 88](#_Toc31567257)

# License information

The MIT License (MIT)

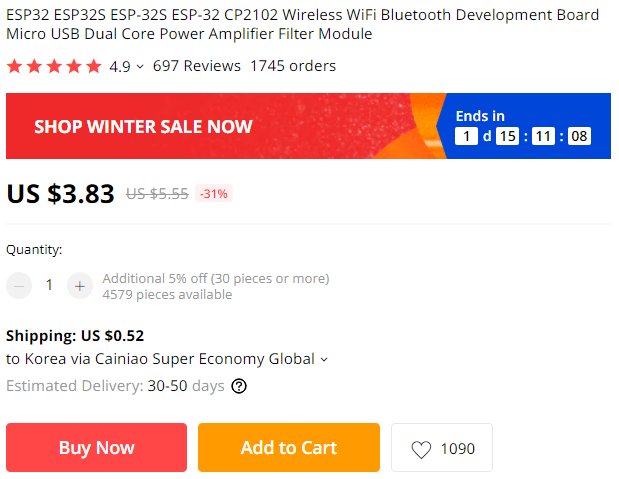
Copyright (c) 2013-2017 Damien P. George, and others

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

**ESP32S ESP-32S ESP32 ESP-32 CP2102 Wireless WiFi Bluetooth Development Board Micro USB Dual Core Power Amplifier Filter Module**

ESP32 is already integrated antenna and RF balun, power amplifier, low-noise amplifiers, filters, and power management module.  
The entire solution takes up the least amount of printed circuit board area.  
This board is used with 2.4 GHz dual-mode Wi-Fi and Bluetooth chips by TSMC 40nm low power technology, power and RF properties best, which is safe, reliable, and scalable to a variety of applications.  
  
**Features:**  
High performance-price ratio  
Small volume, easily embeded to other products   
Strong function with support LWIP protocol, Freertos  
Supporting three modes: AP, STA, and AP+STA  
Supporting Lua program, easily to develop  
  
**Development Board:**  
https://github.com/espressif/arduino-esp32  
https://github.com/nodemcu/nodemcu-firmware/tree/dev-esp32  
https://github.com/Nicholas3388/LuaNode  
  
**If it cann't get bluetooth working?**  
Using Node32s as the Board selection. To reference GPIO pins in code use just the number, for example "digitalWrite(13, HIGH)" sets GPIO13 high.  
The built-in LED to GPIO2.  
  
**Package including:**  
1 x ESP32 Development Board

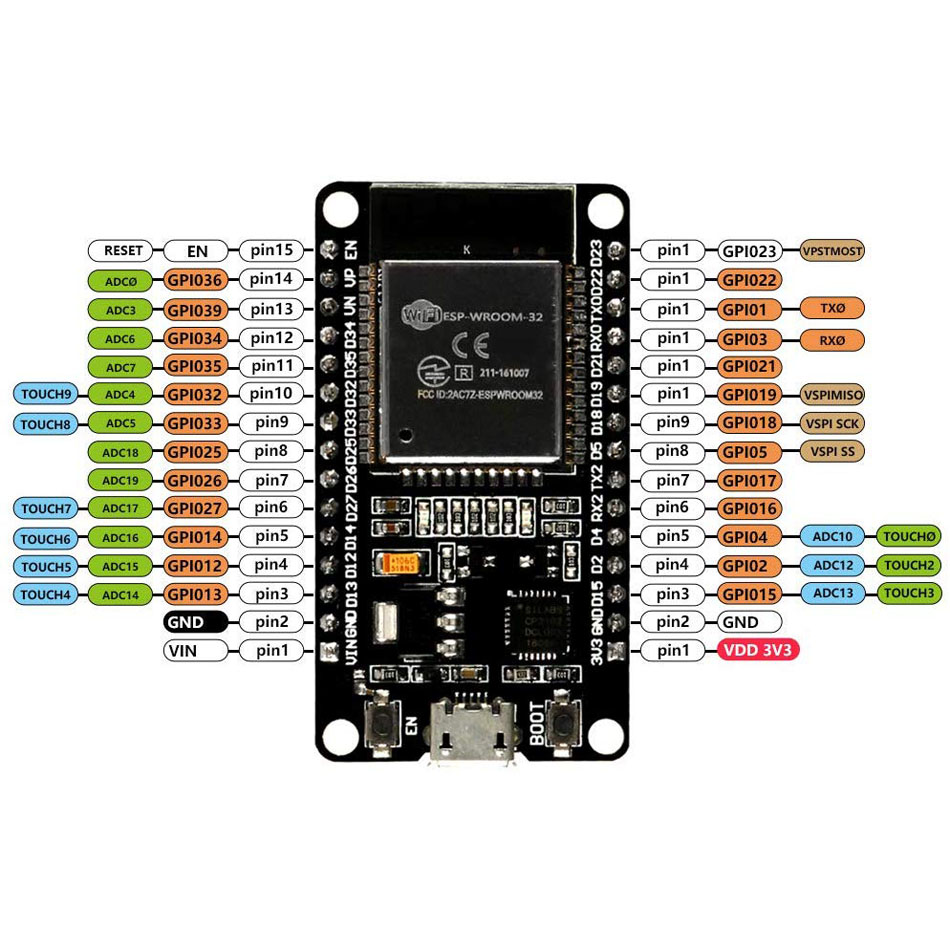
<http://docs.micropython.org/en/latest/>

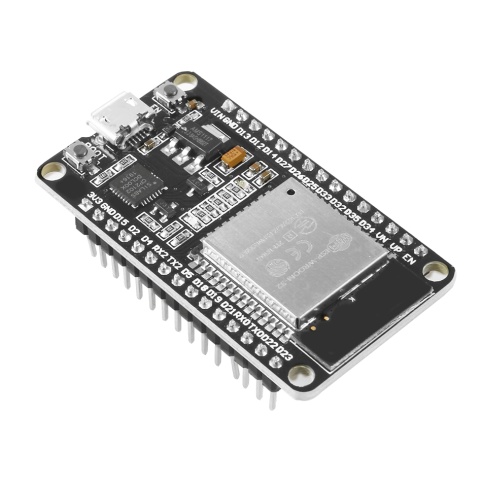
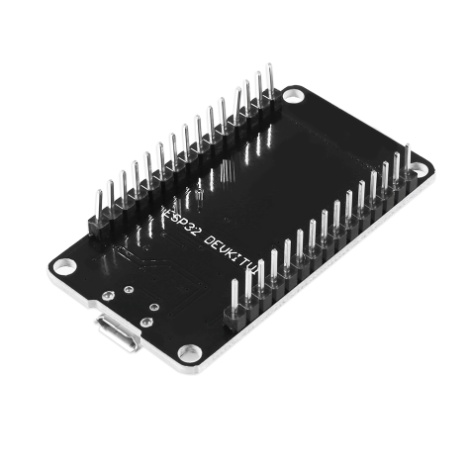
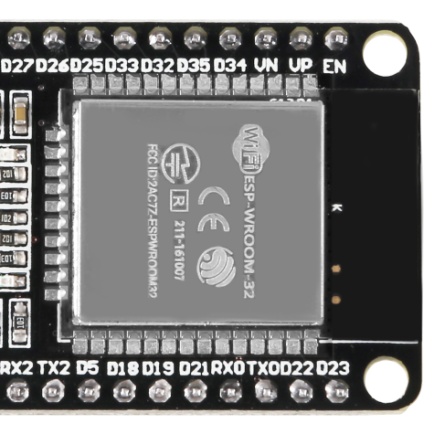
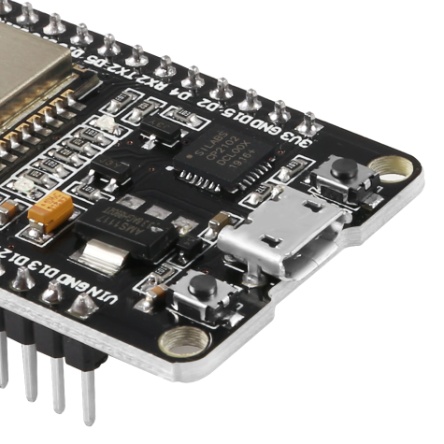
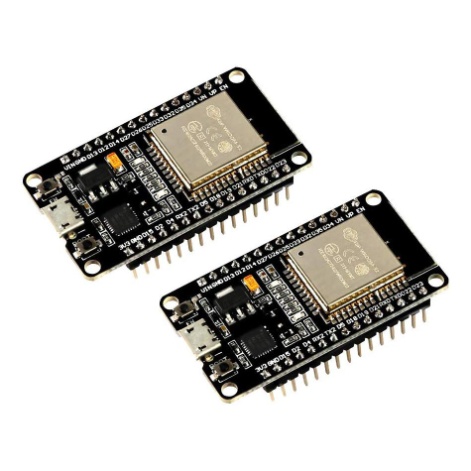
<https://learn.sparkfun.com/tutorials/micropython-programming-tutorial-getting-started-with-the-esp32-thing/resources-and-going-further>

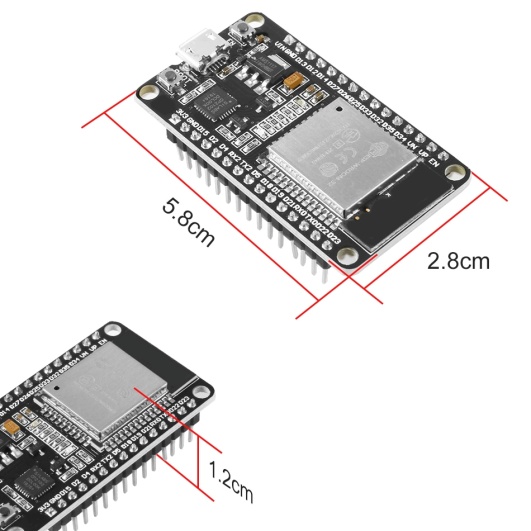
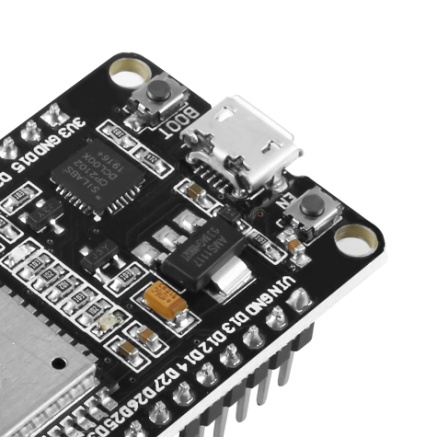
**<Note>**

In nizhny novgorod the parcel came in 10 days. Works fine!

**Attention!!** During downloading the firmware you need to press the boot button, otherwise it does not flash. If the board does not connect to the computer, do not rush to throw it out. Try to change the cable to connect. Need a normal cable, not the one that has the charging function







# Quick reference for the ESP32

Below is a quick reference for ESP32-based boards.

If it is your first time working with this board it may be useful to get an overview of the microcontroller:

* [General information about the ESP32 port](https://docs.micropython.org/en/latest/esp32/general.html)
* [Getting started with MicroPython on the ESP32](https://docs.micropython.org/en/latest/esp32/tutorial/intro.html)

**Installing MicroPython**

See the corresponding section of tutorial: [Getting started with MicroPython on the ESP32](https://docs.micropython.org/en/latest/esp32/tutorial/intro.html#esp32-intro).

It also includes a troubleshooting subsection.

## General board control

The MicroPython REPL is on UART0 (GPIO1=TX, GPIO3=RX) at baudrate 115200. Tab-completion is useful to find out what methods an object has. Paste mode (ctrl-E) is useful to paste a large slab of Python code into the REPL.

The [**machine**](https://docs.micropython.org/en/latest/library/machine.html#module-machine) module:

**import** **machine**

machine.freq() *# get the current frequency of the CPU*

machine.freq(240000000) *# set the CPU frequency to 240 MHz*

The [esp](https://docs.micropython.org/en/latest/library/esp.html#module-esp) module:

**import** **esp**

esp.osdebug(**None**) *# turn off vendor O/S debugging messages*

esp.osdebug(0) *# redirect vendor O/S debugging messages to UART(0)*

*# low level methods to interact with flash storage*

esp.flash\_size()

esp.flash\_user\_start()

esp.flash\_erase(sector\_no)

esp.flash\_write(byte\_offset, buffer)

esp.flash\_read(byte\_offset, buffer)

The [esp32](https://docs.micropython.org/en/latest/library/esp32.html#module-esp32) module:

**import** **esp32**

esp32.hall\_sensor() *# read the internal hall sensor*

esp32.raw\_temperature() *# read the internal temperature of the MCU, in Farenheit*

esp32.ULP() *# access to the Ultra-Low-Power Co-processor*

Note that the temperature sensor in the ESP32 will typically read higher than ambient due to the IC getting warm while it runs. This effect can be minimized by reading the temperature sensor immediately after waking up from sleep.

## Networking

The [network](https://docs.micropython.org/en/latest/library/network.html#module-network) module:

**import** **network**

wlan = network.WLAN(network.STA\_IF) *# create station interface*

wlan.active(**True**) *# activate the interface*

wlan.scan() *# scan for access points*

wlan.isconnected() *# check if the station is connected to an AP*

wlan.connect('essid', 'password') *# connect to an AP*

wlan.config('mac') *# get the interface's MAC address*

wlan.ifconfig() *# get the interface's IP/netmask/gw/DNS addresses*

ap = network.WLAN(network.AP\_IF) *# create access-point interface*

ap.config(essid='ESP-AP') *# set the ESSID of the access point*

ap.active(**True**) *# activate the interface*

A useful function for connecting to your local WiFi network is:

**def** do\_connect():

**import** **network**

wlan = network.WLAN(network.STA\_IF)

wlan.active(**True**)

**if** **not** wlan.isconnected():

print('connecting to network...')

wlan.connect('essid', 'password')

**while** **not** wlan.isconnected():

**pass**

print('network config:', wlan.ifconfig())

Once the network is established the [**socket**](https://docs.micropython.org/en/latest/library/usocket.html#module-usocket) module can be used to create and use TCP/UDP sockets as usual, and the urequests module for convenient HTTP requests.

## Delay and timing

Use the [time](https://docs.micropython.org/en/latest/library/utime.html#module-utime) module:

**import** **time**

time.sleep(1) *# sleep for 1 second*

time.sleep\_ms(500) *# sleep for 500 milliseconds*

time.sleep\_us(10) *# sleep for 10 microseconds*

start = time.ticks\_ms() *# get millisecond counter*

delta = time.ticks\_diff(time.ticks\_ms(), start) *# compute time difference*

## Timers

Virtual (RTOS-based) timers are supported. Use the [machine.Timer](https://docs.micropython.org/en/latest/library/machine.Timer.html#machine-timer) class with timer ID of -1:

**from** **machine** **import** Timer

tim = Timer(-1)

tim.init(period=5000, mode=Timer.ONE\_SHOT, callback=**lambda** t:print(1))

tim.init(period=2000, mode=Timer.PERIODIC, callback=**lambda** t:print(2))

The period is in milliseconds.

## Pins and GPIO

Use the [machine.Pin](https://docs.micropython.org/en/latest/library/machine.Pin.html#machine-pin) class:

**from** **machine** **import** Pin

p0 = Pin(0, Pin.OUT) *# create output pin on GPIO0*

p0.on() *# set pin to "on" (high) level*

p0.off() *# set pin to "off" (low) level*

p0.value(1) *# set pin to on/high*

p2 = Pin(2, Pin.IN) *# create input pin on GPIO2*

print(p2.value()) *# get value, 0 or 1*

p4 = Pin(4, Pin.IN, Pin.PULL\_UP) *# enable internal pull-up resistor*

p5 = Pin(5, Pin.OUT, value=1) *# set pin high on creation*

Available Pins are from the following ranges (inclusive): 0-19, 21-23, 25-27, 32-39. These correspond to the actual GPIO pin numbers of ESP32 chip. Note that many end-user boards use their own adhoc pin numbering (marked e.g. D0, D1, ...). For mapping between board logical pins and physical chip pins consult your board documentation.

Notes:

* Pins 1 and 3 are REPL UART TX and RX respectively
* Pins 6, 7, 8, 11, 16, and 17 are used for connecting the embedded flash, and are not recommended for other uses
* Pins 34-39 are input only, and also do not have internal pull-up resistors
* The pull value of some pins can be set to Pin.PULL\_HOLD to reduce power consumption during deepsleep.

## PWM (pulse width modulation)

PWM can be enabled on all output-enabled pins. The base frequency can range from 1Hz to 40MHz but there is a tradeoff; as the base frequency *increases* the duty resolution *decreases*. See [LED Control](https://docs.espressif.com/projects/esp-idf/en/latest/api-reference/peripherals/ledc.html) for more details.

Use the machine.PWM class:

**from** **machine** **import** Pin, PWM

pwm0 = PWM(Pin(0)) *# create PWM object from a pin*

pwm0.freq() *# get current frequency*

pwm0.freq(1000) *# set frequency*

pwm0.duty() *# get current duty cycle*

pwm0.duty(200) *# set duty cycle*

pwm0.deinit() *# turn off PWM on the pin*

pwm2 = PWM(Pin(2), freq=20000, duty=512) *# create and configure in one go*

## ADC (analog to digital conversion)

On the ESP32 ADC functionality is available on Pins 32-39. Note that, when using the default configuration, input voltages on the ADC pin must be between 0.0v and 1.0v (anything above 1.0v will just read as 4095). Attenuation must be applied in order to increase this usable voltage range.

Use the [machine.ADC](https://docs.micropython.org/en/latest/library/machine.ADC.html#machine-adc) class:

**from** **machine** **import** ADC

adc = ADC(Pin(32)) *# create ADC object on ADC pin*

adc.read() *# read value, 0-4095 across voltage range 0.0v - 1.0v*

adc.atten(ADC.ATTN\_11DB) *# set 11dB input attenuation (voltage range roughly 0.0v - 3.6v)*

adc.width(ADC.WIDTH\_9BIT) *# set 9 bit return values (returned range 0-511)*

adc.read() *# read value using the newly configured attenuation and width*

ESP32 specific ADC class method reference:

**ADC.atten(*attenuation*)**

This method allows for the setting of the amount of attenuation on the input of the ADC. This allows for a wider possible input voltage range, at the cost of accuracy (the same number of bits now represents a wider range). The possible attenuation options are:

* **ADC.ATTN\_0DB**: 0dB attenuation, gives a maximum input voltage of 1.00v - this is the default configuration
* **ADC.ATTN\_2\_5DB**: 2.5dB attenuation, gives a maximum input voltage of approximately 1.34v
* **ADC.ATTN\_6DB**: 6dB attenuation, gives a maximum input voltage of approximately 2.00v
* **ADC.ATTN\_11DB**: 11dB attenuation, gives a maximum input voltage of approximately 3.6v

**Warning**

Despite 11dB attenuation allowing for up to a 3.6v range, note that the absolute maximum voltage rating for the input pins is 3.6v, and so going near this boundary may be damaging to the IC!

**ADC.width(*width*)**

This method allows for the setting of the number of bits to be utilised and returned during ADC reads. Possible width options are:

* **ADC.WIDTH\_9BIT**: 9 bit data
* **ADC.WIDTH\_10BIT**: 10 bit data
* **ADC.WIDTH\_11BIT**: 11 bit data
* **ADC.WIDTH\_12BIT**: 12 bit data - this is the default configuration

## Software SPI bus

There are two SPI drivers. One is implemented in software (bit-banging) and works on all pins, and is accessed via the [machine.SPI](https://docs.micropython.org/en/latest/library/machine.SPI.html#machine-spi) class:

**from** **machine** **import** Pin, SPI

*# construct an SPI bus on the given pins*

*# polarity is the idle state of SCK*

*# phase=0 means sample on the first edge of SCK, phase=1 means the second*

spi = SPI(baudrate=100000, polarity=1, phase=0, sck=Pin(0), mosi=Pin(2), miso=Pin(4))

spi.init(baudrate=200000) *# set the baudrate*

spi.read(10) *# read 10 bytes on MISO*

spi.read(10, 0xff) *# read 10 bytes while outputting 0xff on MOSI*

buf = bytearray(50) *# create a buffer*

spi.readinto(buf) *# read into the given buffer (reads 50 bytes in this case)*

spi.readinto(buf, 0xff) *# read into the given buffer and output 0xff on MOSI*

spi.write(b'12345') *# write 5 bytes on MOSI*

buf = bytearray(4) *# create a buffer*

spi.write\_readinto(b'1234', buf) *# write to MOSI and read from MISO into the buffer*

spi.write\_readinto(buf, buf) *# write buf to MOSI and read MISO back into buf*

**Warning**

Currently *all* of sck, mosi and miso *must* be specified when initialising Software SPI.

## Hardware SPI bus

There are two hardware SPI channels that allow faster transmission rates (up to 80Mhz). These may be used on any IO pins that support the required direction and are otherwise unused (see [Pins and GPIO](https://docs.micropython.org/en/latest/esp32/quickref.html#pins-and-gpio)) but if they are not configured to their default pins then they need to pass through an extra layer of GPIO multiplexing, which can impact their reliability at high speeds. Hardware SPI channels are limited to 40MHz when used on pins other than the default ones listed below.

|  | **HSPI (id=1)** | **VSPI (id=2)** |
| --- | --- | --- |
| **sck** | 14 | 18 |
| **mosi** | 13 | 23 |
| **miso** | 12 | 19 |

Hardware SPI has the same methods as Software SPI above:

**from** **machine** **import** Pin, SPI

hspi = SPI(1, 10000000, sck=Pin(14), mosi=Pin(13), miso=Pin(12))

vspi = SPI(2, baudrate=80000000, polarity=0, phase=0, bits=8, firstbit=0, sck=Pin(18), mosi=Pin(23), miso=Pin(19))

## I2C bus

The I2C driver has both software and hardware implementations, and the two hardware peripherals have identifiers 0 and 1. Any available output-capable pins can be used for SCL and SDA. The driver is accessed via the [machine.I2C](https://docs.micropython.org/en/latest/library/machine.I2C.html#machine-i2c) class:

**from** **machine** **import** Pin, I2C

*# construct a software I2C bus*

i2c = I2C(scl=Pin(5), sda=Pin(4), freq=100000)

*# construct a hardware I2C bus*

i2c = I2C(0)

i2c = I2C(1, scl=Pin(5), sda=Pin(4), freq=400000)

i2c.scan() *# scan for slave devices*

i2c.readfrom(0x3a, 4) *# read 4 bytes from slave device with address 0x3a*

i2c.writeto(0x3a, '12') *# write '12' to slave device with address 0x3a*

buf = bytearray(10) *# create a buffer with 10 bytes*

i2c.writeto(0x3a, buf) *# write the given buffer to the slave*

## Real time clock (RTC)

See [machine.RTC](https://docs.micropython.org/en/latest/library/machine.RTC.html#machine-rtc)

**from** **machine** **import** RTC

rtc = RTC()

rtc.datetime((2017, 8, 23, 1, 12, 48, 0, 0)) *# set a specific date and time*

rtc.datetime() *# get date and time*

## Deep-sleep mode

The following code can be used to sleep, wake and check the reset cause:

**import** **machine**

*# check if the device woke from a deep sleep*

**if** machine.reset\_cause() == machine.DEEPSLEEP\_RESET:

print('woke from a deep sleep')

*# put the device to sleep for 10 seconds*

machine.deepsleep(10000)

Notes:

Calling deepsleep() without an argument will put the device to sleep indefinitely

* A software reset does not change the reset cause
* There may be some leakage current flowing through enabled internal pullups. To further reduce power consumption it is possible to disable the internal pullups:
* p1 = Pin(4, Pin.IN, Pin.PULL\_HOLD)

After leaving deepsleep it may be necessary to un-hold the pin explicitly (e.g. if it is an output pin) via:

* p1 = Pin(4, Pin.OUT, **None**)

## RMT

The RMT is ESP32-specific and allows generation of accurate digital pulses with 12.5ns resolution. See [esp32.RMT](https://docs.micropython.org/en/latest/library/esp32.html#esp32-rmt) for details. Usage is:

**import** **esp32**

**from** **machine** **import** Pin

r = esp32.RMT(0, pin=Pin(18), clock\_div=8)

r *# RMT(channel=0, pin=18, source\_freq=80000000, clock\_div=8)*

*# The channel resolution is 100ns (1/(source\_freq/clock\_div)).*

r.write\_pulses((1, 20, 2, 40), start=0) *# Send 0 for 100ns, 1 for 2000ns, 0 for 200ns, 1 for 4000ns*

## OneWire driver

The OneWire driver is implemented in software and works on all pins:

**from** **machine** **import** Pin

**import** **onewire**

ow = onewire.OneWire(Pin(12)) *# create a OneWire bus on GPIO12*

ow.scan() *# return a list of devices on the bus*

ow.reset() *# reset the bus*

ow.readbyte() *# read a byte*

ow.writebyte(0x12) *# write a byte on the bus*

ow.write('123') *# write bytes on the bus*

ow.select\_rom(b'12345678') *# select a specific device by its ROM code*

There is a specific driver for DS18S20 and DS18B20 devices:

**import** **time**, **ds18x20**

ds = ds18x20.DS18X20(ow)

roms = ds.scan()

ds.convert\_temp()

time.sleep\_ms(750)

**for** rom **in** roms:

print(ds.read\_temp(rom))

Be sure to put a 4.7k pull-up resistor on the data line. Note that the convert\_temp() method must be called each time you want to sample the temperature.

## NeoPixel driver

Use the neopixel module:

**from** **machine** **import** Pin

**from** **neopixel** **import** NeoPixel

pin = Pin(0, Pin.OUT) *# set GPIO0 to output to drive NeoPixels*

np = NeoPixel(pin, 8) *# create NeoPixel driver on GPIO0 for 8 pixels*

np[0] = (255, 255, 255) *# set the first pixel to white*

np.write() *# write data to all pixels*

r, g, b = np[0] *# get first pixel colour*

For low-level driving of a NeoPixel:

**import** **esp**

esp.neopixel\_write(pin, grb\_buf, is800khz)

**Warning**

By default NeoPixel is configured to control the more popular *800kHz* units. It is possible to use alternative timing to control other (typically 400kHz) devices by passing timing=0 when constructing the NeoPixel object.

## Capacitive touch

Use the TouchPad class in the machine module:

**from** **machine** **import** TouchPad, Pin

t = TouchPad(Pin(14))

t.read() *# Returns a smaller number when touched*

TouchPad.read returns a value relative to the capacitive variation. Small numbers (typically in the *tens*) are common when a pin is touched, larger numbers (above *one thousand*) when no touch is present. However the values are *relative* and can vary depending on the board and surrounding composition so some calibration may be required.

There are ten capacitive touch-enabled pins that can be used on the ESP32: 0, 2, 4, 12, 13 14, 15, 27, 32, 33. Trying to assign to any other pins will result in a ValueError.

Note that TouchPads can be used to wake an ESP32 from sleep:

**import** **machine**

**from** **machine** **import** TouchPad, Pin

**import** **esp32**

t = TouchPad(Pin(14))

t.config(500) *# configure the threshold at which the pin is considered touched*

esp32.wake\_on\_touch(**True**)

machine.lightsleep() *# put the MCU to sleep until a touchpad is touched*

For more details on touchpads refer to [Espressif Touch Sensor](https://docs.espressif.com/projects/esp-idf/en/latest/api-reference/peripherals/touch_pad.html).

## DHT driver

The DHT driver is implemented in software and works on all pins:

**import** **dht**

**import** **machine**

d = dht.DHT11(machine.Pin(4))

d.measure()

d.temperature() *# eg. 23 (°C)*

d.humidity() *# eg. 41 (% RH)*

d = dht.DHT22(machine.Pin(4))

d.measure()

d.temperature() *# eg. 23.6 (°C)*

d.humidity() *# eg. 41.3 (% RH)*

## WebREPL (web browser interactive prompt)

WebREPL (REPL over WebSockets, accessible via a web browser) is an experimental feature available in ESP32 port. Download web client from <https://github.com/micropython/webrepl> (hosted version available at [http://micropython.org/ webrepl](http://micropython.org/webrepl)), and configure it by executing:

**import** **webrepl\_setup**

and following on-screen instructions. After reboot, it will be available for connection. If you disabled automatic start-up on boot, you may run configured daemon on demand using:

**import** **webrepl**

webrepl.start()

*# or, start with a specific password*

webrepl.start(password='mypass')

The WebREPL daemon listens on all active interfaces, which can be STA or AP. This allows you to connect to the ESP32 via a router (the STA interface) or directly when connected to its access point.

In addition to terminal/command prompt access, WebREPL also has provision for file transfer (both upload and download). The web client has buttons for the corresponding functions, or you can use the command-line client webrepl\_cli.py from the repository above.

See the MicroPython forum for other community-supported alternatives to transfer files to an ESP32 board.

# MicroPython language and implementation

MicroPython aims to implement the Python 3.4 standard (with selected features from later versions) with respect to language syntax, and most of the features of MicroPython are identical to those described by the “Language Reference” documentation at [docs.python.org](https://docs.python.org/3/reference/index.html). The MicroPython standard library is described in the [corresponding chapter](https://docs.micropython.org/en/latest/library/index.html#micropython-lib). The [MicroPython differences from CPython](https://docs.micropython.org/en/latest/genrst/index.html#cpython-diffs) chapter describes differences between MicroPython and CPython (which mostly concern standard library and types, but also some language-level features).

This chapter describes features and peculiarities of MicroPython implementation and the best practices to use them.

* [Glossary](https://docs.micropython.org/en/latest/reference/glossary.html)
* [The MicroPython Interactive Interpreter Mode (aka REPL)](https://docs.micropython.org/en/latest/reference/repl.html)
* [MicroPython .mpy files](https://docs.micropython.org/en/latest/reference/mpyfiles.html)
* [Writing interrupt handlers](https://docs.micropython.org/en/latest/reference/isr_rules.html)
* [Maximising MicroPython speed](https://docs.micropython.org/en/latest/reference/speed_python.html)
* [MicroPython on microcontrollers](https://docs.micropython.org/en/latest/reference/constrained.html)
* [Distribution packages, package management, and deploying applications](https://docs.micropython.org/en/latest/reference/packages.html)
* [Inline assembler for Thumb2 architectures](https://docs.micropython.org/en/latest/reference/asm_thumb2_index.html)
* [Working with filesystems](https://docs.micropython.org/en/latest/reference/filesystem.html)
* [The pyboard.py tool](https://docs.micropython.org/en/latest/reference/pyboard.py.html)

# Glossary

**baremetal**

A system without a (full-fledged) operating system, for example an [MCU](https://docs.micropython.org/en/latest/reference/glossary.html#term-mcu)-based system. When running on a baremetal system, MicroPython effectively functions like a small operating system, running user programs and providing a command interpreter ([REPL](https://docs.micropython.org/en/latest/reference/glossary.html#term-repl)).

**buffer protocol**

Any Python object that can be automatically converted into bytes, such as bytes, bytearray, memoryview and str objects, which all implement the “buffer protocol”.

**board**

Typically this refers to a printed circuit board (PCB) containing a [microcontroller](https://docs.micropython.org/en/latest/reference/glossary.html#term-mcu) and supporting components. MicroPython firmware is typically provided per-board, as the firmware contains both MCU-specific functionality but also board-level functionality such as drivers or pin names.

**bytecode**

A compact representation of a Python program that generated by compiling the Python source code. This is what the VM actually executes. Bytecode is typically generated automatically at runtime and is invisible to the user. Note that while [CPython](https://docs.micropython.org/en/latest/reference/glossary.html#term-cpython) and MicroPython both use bytecode, the format is different. You can also pre-compile source code offline using the [cross-compiler](https://docs.micropython.org/en/latest/reference/glossary.html#term-cross-compiler).

**callee-owned tuple**

This is a MicroPython-specific construct where, for efficiency reasons, some built-in functions or methods may re-use the same underlying tuple object to return data. This avoids having to allocate a new tuple for every call, and reduces heap fragmentation. Programs should not hold references to callee-owned tuples and instead only extract data from them (or make a copy).

**CircuitPython**

A variant of MicroPython developed by [Adafruit Industries](https://circuitpython.org/).

**CPython**

CPython is the reference implementation of the Python programming language, and the most well-known one. It is, however, one of many implementations (including Jython, IronPython, PyPy, and MicroPython). While MicroPython’s implementation differs substantially from CPython, it aims to maintain as much compatibility as possible.

**cross-compiler**

Also known as mpy-cross. This tool runs on your PC and converts a [.py file](https://docs.micropython.org/en/latest/reference/glossary.html#term-py-file) containing MicroPython code into a [.mpy file](https://docs.micropython.org/en/latest/reference/glossary.html#term-mpy-file) containing MicroPython bytecode. This means it loads faster (the board doesn’t have to compile the code), and uses less space on flash (the bytecode is more space efficient).

**driver**

A MicroPython library that implements support for a particular component, such as a sensor or display.

**FFI**

Acronym for Foreign Function Interface. A mechanism used by the [MicroPython Unix port](https://docs.micropython.org/en/latest/reference/glossary.html#term-micropython-unix-port) to access operating system functionality. This is not available on [baremetal](https://docs.micropython.org/en/latest/reference/glossary.html#term-baremetal) ports.

**filesystem**

Most MicroPython ports and boards provide a filesystem stored in flash that is available to user code via the standard Python file APIs such as open(). Some boards also make this internal filesystem accessible to the host via USB mass-storage.

**frozen module**

A Python module that has been cross compiled and bundled into the firmware image. This reduces RAM requirements as the code is executed directly from flash.

**Garbage Collector**

A background process that runs in Python (and MicroPython) to reclaim unused memory in the [heap](https://docs.micropython.org/en/latest/reference/glossary.html#term-heap).

**GPIO**

General-purpose input/output. The simplest means to control electrical signals (commonly referred to as “pins”) on a microcontroller. GPIO typically allows pins to be either input or output, and to set or get their digital value (logical “0” or “1”). MicroPython abstracts GPIO access using the [machine.Pin](https://docs.micropython.org/en/latest/library/machine.Pin.html#machine.Pin) and [machine.Signal](https://docs.micropython.org/en/latest/library/machine.Signal.html#machine.Signal) classes.

**GPIO port**

A group of [GPIO](https://docs.micropython.org/en/latest/reference/glossary.html#term-gpio) pins, usually based on hardware properties of these pins (e.g. controllable by the same register).

**heap**

A region of RAM where MicroPython stores dynamic data. It is managed automatically by the [Garbage Collector](https://docs.micropython.org/en/latest/reference/glossary.html#term-garbage-collector). Different MCUs and boards have vastly different amounts of RAM available for the heap, so this will affect how complex your program can be.

**interned string**

An optimisation used by MicroPython to improve the efficiency of working with strings. An interned string is referenced by its (unique) identity rather than its address and can therefore be quickly compared just by its identifier. It also means that identical strings can be de-duplicated in memory. String interning is almost always invisible to the user.

**MCU**

Microcontroller. Microcontrollers usually have much less resources than a desktop, laptop, or phone, but are smaller, cheaper and require much less power. MicroPython is designed to be small and optimized enough to run on an average modern microcontroller.

**micropython-lib**

MicroPython is (usually) distributed as a single executable/binary file with just few builtin modules. There is no extensive standard library comparable with [CPython](https://docs.micropython.org/en/latest/reference/glossary.html#term-cpython)‘s. Instead, there is a related, but separate project [micropython-lib](https://github.com/micropython/micropython-lib) which provides implementations for many modules from CPython’s standard library.

Some of the modules are are implemented in pure Python, and are able to be used on all ports. However, the majority of these modules use [FFI](https://docs.micropython.org/en/latest/reference/glossary.html#term-ffi) to access operating system functionality, and as such can only be used on the [MicroPython Unix port](https://docs.micropython.org/en/latest/reference/glossary.html#term-micropython-unix-port) (with limited support for Windows).

Unlike the [CPython](https://docs.micropython.org/en/latest/reference/glossary.html#term-cpython) stdlib, micropython-lib modules are intended to be installed individually - either using manual copying or using [upip](https://docs.micropython.org/en/latest/reference/glossary.html#term-upip).

**MicroPython port**

MicroPython supports different [boards](https://docs.micropython.org/en/latest/reference/glossary.html#term-board), RTOSes, and OSes, and can be relatively easily adapted to new systems. MicroPython with support for a particular system is called a “port” to that system. Different ports may have widely different functionality. This documentation is intended to be a reference of the generic APIs available across different ports (“MicroPython core”). Note that some ports may still omit some APIs described here (e.g. due to resource constraints). Any such differences, and port-specific extensions beyond the MicroPython core functionality, would be described in the separate port-specific documentation.

**MicroPython Unix port**

The unix port is one of the major [MicroPython ports](https://docs.micropython.org/en/latest/reference/glossary.html#term-micropython-port). It is intended to run on POSIX-compatible operating systems, like Linux, MacOS, FreeBSD, Solaris, etc. It also serves as the basis of Windows port. The Unix port is very useful for quick development and testing of the MicroPython language and machine-independent features. It can also function in a similar way to [CPython](https://docs.micropython.org/en/latest/reference/glossary.html#term-cpython)‘s python executable.

**.mpy file**

The output of the [cross-compiler](https://docs.micropython.org/en/latest/reference/glossary.html#term-cross-compiler). A compiled form of a [.py file](https://docs.micropython.org/en/latest/reference/glossary.html#term-py-file) that contains MicroPython bytecode instead of Python source code.

**native**

Usually refers to “native code”, i.e. machine code for the target microcontroller (such as ARM Thumb, Xtensa, x86/x64). The @native decorator can be applied to a MicroPython function to generate native code instead of bytecode for that function, which will likely be faster but use more RAM.

**port**

Usually short for [MicroPython port](https://docs.micropython.org/en/latest/reference/glossary.html#term-micropython-port), but could also refer to [GPIO port](https://docs.micropython.org/en/latest/reference/glossary.html#term-gpio-port).

**.py file**

A file containing Python source code.

**REPL**

An acronym for “Read, Eval, Print, Loop”. This is the interactive Python prompt, useful for debugging or testing short snippets of code. Most MicroPython boards make a REPL available over a UART, and this is typically accessible on a host PC via USB.

**stream**

Also known as a “file-like object”. An Python object which provides sequential read-write access to the underlying data. A stream object implements a corresponding interface, which consists of methods like read(),  write(), readinto(),  seek(),

flush(), close(), etc. A stream is an important concept in MicroPython; many I/O objects implement the stream interface, and thus can be used consistently and interchangeably in different contexts. For more information on streams in MicroPython, see the [uio](https://docs.micropython.org/en/latest/library/uio.html#module-uio) module.

**UART**

Acronym for “Universal Asynchronous Receiver/Transmitter”. This is a peripheral that sends data over a pair of pins (TX & RX). Many boards include a way to make at least one of the UARTs available to a host PC as a serial port over USB.

**upip**

(Literally, “micro pip”). A package manager for MicroPython, inspired by [CPython](https://docs.micropython.org/en/latest/reference/glossary.html#term-cpython)‘s pip, but much smaller and with reduced functionality. upip runs both on the [Unix port](https://docs.micropython.org/en/latest/reference/glossary.html#term-micropython-unix-port) and on [baremetal](https://docs.micropython.org/en/latest/reference/glossary.html#term-baremetal) ports which offer filesystem and networking support.

# The MicroPython Interactive Interpreter Mode (aka REPL)

This section covers some characteristics of the MicroPython Interactive Interpreter Mode. A commonly used term for this is REPL (read-eval-print-loop) which will be used to refer to this interactive prompt.

## Auto-indent

When typing python statements which end in a colon (for example if, for, while) then the prompt will change to three dots (...) and the cursor will be indented by 4 spaces. When you press return, the next line will continue at the same level of indentation for regular statements or an additional level of indentation where appropriate. If you press the backspace key then it will undo one level of indentation.

If your cursor is all the way back at the beginning, pressing RETURN will then execute the code that you’ve entered. The following shows what you’d see after entering a for statement (the underscore shows where the cursor winds up):

**>>> for** i **in** range(30):

**...**  \_

If you then enter an if statement, an additional level of indentation will be provided:

**>>> for** i **in** range(30):

**...**  **if** i > 3:

**...**  \_

Now enter break followed by RETURN and press BACKSPACE:

**>>> for** i **in** range(30):

**...**  **if** i > 3:

**...**  **break**

**...**  \_

Finally type print(i), press RETURN, press BACKSPACE and press RETURN again:

**>>> for** i **in** range(30):

**...**  **if** i > 3:

**...**  **break**

**...**  print(i)

**...**

0

1

2

3

>>>

Auto-indent won’t be applied if the previous two lines were all spaces. This means that you can finish entering a compound statement by pressing RETURN twice, and then a third press will finish and execute.

## Auto-completion

While typing a command at the REPL, if the line typed so far corresponds to the beginning of the name of something, then pressing TAB will show possible things that could be entered. For example, first import the machine module by entering import machine and pressing RETURN. Then type m and press TAB and it should expand to machine. Enter a dot . and press TAB again. You should see something like:

**>>>** machine.

\_\_name\_\_ info unique\_id reset

bootloader freq rng idle

sleep deepsleep disable\_irq enable\_irq

Pin

The word will be expanded as much as possible until multiple possibilities exist. For example, type machine.Pin.AF3 and press TAB and it will expand to machine.Pin.AF3\_TIM. Pressing TAB a second time will show the possible expansions:

**>>>** machine.Pin.AF3\_TIM

AF3\_TIM10 AF3\_TIM11 AF3\_TIM8 AF3\_TIM9

**>>>** machine.Pin.AF3\_TIM

## Interrupting a running program

You can interrupt a running program by pressing Ctrl-C. This will raise a KeyboardInterrupt which will bring you back to the REPL, providing your program doesn’t intercept the KeyboardInterrupt exception.

For example:

**>>> for** i **in** range(1000000):

**...**  print(i)

**...**

0

1

2

3

**...**

6466

6467

6468

Traceback (most recent call last):

File "<stdin>", line 2, in <module>

KeyboardInterrupt:

>>>

## Paste mode

If you want to paste some code into your terminal window, the auto-indent feature will mess things up. For example, if you had the following python code:

**def** foo():

print('This is a test to show paste mode')

print('Here is a second line')

foo()

and you try to paste this into the normal REPL, then you will see something like this:

**>>> def** foo():

**...**  print('This is a test to show paste mode')

**...**  print('Here is a second line')

**...**  foo()

**...**

File "<stdin>", line 3

IndentationError: unexpected indent

If you press Ctrl-E, then you will enter paste mode, which essentially turns off the auto-indent feature, and changes the prompt from >>> to ===. For example:

>>>

paste mode; Ctrl-C to cancel, Ctrl-D to finish

=== def foo():

=== print('This is a test to show paste mode')

=== print('Here is a second line')

=== foo()

===

This is a test to show paste mode

Here is a second line

>>>

Paste Mode allows blank lines to be pasted. The pasted text is compiled as if it were a file. Pressing Ctrl-D exits paste mode and initiates the compilation.

## Soft reset

A soft reset will reset the python interpreter, but tries not to reset the method by which you’re connected to the MicroPython board (USB-serial, or Wifi).

You can perform a soft reset from the REPL by pressing Ctrl-D, or from your python code by executing:

machine.soft\_reset()

For example, if you reset your MicroPython board, and you execute a dir() command, you’d see something like this:

**>>>** dir()

['\_\_name\_\_', 'pyb']

Now create some variables and repeat the dir() command:

**>>>** i = 1

**>>>** j = 23

**>>>** x = 'abc'

**>>>** dir()

['j', 'x', '\_\_name\_\_', 'pyb', 'i']

>>>

Now if you enter Ctrl-D, and repeat the dir() command, you’ll see that your variables no longer exist:

MPY: sync filesystems

MPY: soft reboot

MicroPython v1.5-51-g6f70283-dirty on 2015-10-30; PYBv1.0 **with** STM32F405RG

Type "help()" **for** more information.

>>> dir()

['\_\_name\_\_', 'pyb']

>>>

## The special variable \_ (underscore)

When you use the REPL, you may perform computations and see the results. MicroPython stores the results of the previous statement in the variable \_ (underscore). So you can use the underscore to save the result in a variable. For example:

**>>>** 1 + 2 + 3 + 4 + 5

15

**>>>** x = \_

**>>>** x

15

>>>

## Raw mode

Raw mode is not something that a person would normally use. It is intended for programmatic use. It essentially behaves like paste mode with echo turned off.

Raw mode is entered using Ctrl-A. You then send your python code, followed by a Ctrl-D. The Ctrl-D will be acknowledged by ‘OK’ and then the python code will be compiled and executed. Any output (or errors) will be sent back. Entering Ctrl-B will leave raw mode and return the the regular (aka friendly) REPL.

The tools/pyboard.py program uses the raw REPL to execute python files on the MicroPython board.

# MicroPython .mpy files

MicroPython defines the concept of an .mpy file which is a binary container file format that holds precompiled code, and which can be imported like a normal .py module. The file foo.mpy can be imported via import foo, as long as foo.mpy can be found in the usual way by the import machinery. Usually, each directory listed in sys.path is searched in order. When searching a particular directory foo.py is looked for first and if that is not found then foo.mpy is looked for, then the search continues in the next directory if neither is found. As such, foo.py will take precedence over foo.mpy.

These .mpy files can contain bytecode which is usually generated from Python source files (.py files) via the mpy-cross program. For some architectures an .mpy file can also contain native machine code, which can be generated in a variety of ways, most notably from C source code.

## Versioning and compatibility of .mpy files

A given .mpy file may or may not be compatible with a given MicroPython system. Compatibility is based on the following:

* Version of the .mpy file: the version of the file must match the version supported by the system loading it.
* Bytecode features used in the .mpy file: there are two bytecode features which must match between the file and the system: unicode support and inline caching of map lookups in the bytecode.
* Small integer bits: the .mpy file will require a minimum number of bits in a small integer and the system loading it must support at least this many bits.
* Qstr compression window size: the .mpy file will require a minimum window size for qstr decompression and the system loading it must have a window greater or equal to this size.
* Native architecture: if the .mpy file contains native machine code then it will specify the architecture of that machine code and the system loading it must support execution of that architecture’s code.

If a MicroPython system supports importing .mpy files then the sys.implementation.mpy field will exist and return an integer which encodes the version (lower 8 bits), features and native architecture.

Trying to import an .mpy file that fails one of the first four tests will raise ValueError('incompatible .mpy file'). Trying to import an .mpy file that fails the native architecture test (if it contains native machine code) will raise ValueError('incompatible .mpy arch').

If importing an .mpy file fails then try the following:

Determine the .mpy version and flags supported by your MicroPython system by executing:

**import** **sys**

sys\_mpy = sys.implementation.mpy

arch = [**None**, 'x86', 'x64',

'armv6', 'armv6m', 'armv7m', 'armv7em', 'armv7emsp', 'armv7emdp',

'xtensa', 'xtensawin'][sys\_mpy >> 10]

print('mpy version:', sys\_mpy & 0xff)

print('mpy flags:', end='')

**if** arch:

print(' -march=' + arch, end='')

**if** sys\_mpy & 0x100:

print(' -mcache-lookup-bc', end='')

**if** **not** sys\_mpy & 0x200:

print(' -mno-unicode', end='')

print()

* Check the validity of the .mpy file by inspecting the first two bytes of the file. The first byte should be an uppercase ‘M’ and the second byte will be the version number, which should match the system version from above. If it doesn’t match then rebuild the .mpy file.
* Check if the system .mpy version matches the version emitted by mpy-cross that was used to build the .mpy file, found by mpy-cross --version. If it doesn’t match then recompile mpy-cross from the Git repository checked out at the tag (or hash) reported by mpy-cross --version.
* Make sure you are using the correct mpy-cross flags, found by the code above, or by inspecting the MPY\_CROSS\_FLAGS Makefile variable for the port that you are using.

The following table shows the correspondence between MicroPython release and .mpy version.

| **MicroPython release** | **.mpy version** |
| --- | --- |
| v1.12 and up | 5 |
| v1.11 | 4 |
| v1.9.3 - v1.10 | 3 |
| v1.9 - v1.9.2 | 2 |
| v1.5.1 - v1.8.7 | 0 |

For completeness, the next table shows the Git commit of the main MicroPython repository at which the .mpy version was changed.

| **.mpy version change** | **Git commit** |
| --- | --- |
| 4 to 5 | 5716c5cf65e9b2cb46c2906f40302401bdd27517 |
| 3 to 4 | 9a5f92ea72754c01cc03e5efcdfe94021120531e |
| 2 to 3 | ff93fd4f50321c6190e1659b19e64fef3045a484 |
| 1 to 2 | dd11af209d226b7d18d5148b239662e30ed60bad |
| 0 to 1 | 6a11048af1d01c78bdacddadd1b72dc7ba7c6478 |
| initial version 0 | d8c834c95d506db979ec871417de90b7951edc30 |

## Binary encoding of .mpy files

MicroPython .mpy files are a binary container format with code objects stored internally in a nested hierarchy. To keep files small while still providing a large range of possible values it uses the concept of a variably-encoded-unsigned-integer (vuint) in many places. Similar to utf-8 encoding, this encoding stores 7 bits per byte with the 8th bit (MSB) set if one or more bytes follow. The bits of the unsigned integer are stored in the vuint in LSB form.

The top-level of an .mpy file consists of two parts:

* The header.
* The raw-code for the outer scope of the module. This outer scope is executed when the .mpy file is imported.
* The header

The .mpy header is:

| **size** | **field** |
| --- | --- |
| byte | value 0x4d (ASCII ‘M’) |
| byte | .mpy version number |
| byte | feature flags |
| byte | number of bits in a small int |
| vuint | size of qstr window |

Raw code elements

A raw-code element contains code, either bytecode or native machine code. Its contents are:

| **size** | **field** |
| --- | --- |
| vuint | type and size |
| ... | code (bytecode or machine code) |
| vuint | number of constant objects |
| vuint | number of sub-raw-code elements |
| ... | constant objects |
| ... | sub-raw-code elements |

The first vuint in a raw-code element encodes the type of code stored in this element (the two least-significant bits), and the decompressed length of the code (the amount of RAM to allocate for it).

Following the vuint comes the code itself. In the case of bytecode it also contains compressed qstr values.

Following the code comes a vuint counting the number of constant objects, and another vuint counting the number of sub-raw-code elements.The constant objects are then stored next. Finally any sub-raw-code elements are stored, recursively.

# Writing interrupt handlers

On suitable hardware MicroPython offers the ability to write interrupt handlers in Python. Interrupt handlers - also known as interrupt service routines (ISR’s) - are defined as callback functions. These are executed in response to an event such as a timer trigger or a voltage change on a pin. Such events can occur at any point in the execution of the program code. This carries significant consequences, some specific to the MicroPython language. Others are common to all systems capable of responding to real time events. This document covers the language specific issues first, followed by a brief introduction to real time programming for those new to it.

This introduction uses vague terms like “slow” or “as fast as possible”. This is deliberate, as speeds are application dependent. Acceptable durations for an ISR are dependent on the rate at which interrupts occur, the nature of the main program, and the presence of other concurrent events.

## Tips and recommended practices

This summarises the points detailed below and lists the principal recommendations for interrupt handler code.

* Keep the code as short and simple as possible.
* Avoid memory allocation: no appending to lists or insertion into dictionaries, no floating point.
* Consider using micropython.schedule to work around the above constraint.
* Where an ISR returns multiple bytes use a pre-allocated bytearray. If multiple integers are to be shared between an ISR and the main program consider an array (array.array).
* Where data is shared between the main program and an ISR, consider disabling interrupts prior to accessing the data in the main program and re-enabling them immediately afterwards (see Critical Sections).
* Allocate an emergency exception buffer (see below).

## MicroPython issues

### The emergency exception buffer

If an error occurs in an ISR, MicroPython is unable to produce an error report unless a special buffer is created for the purpose. Debugging is simplified if the following code is included in any program using interrupts.

**import** **micropython**

micropython.alloc\_emergency\_exception\_buf(100)

### Simplicity

For a variety of reasons it is important to keep ISR code as short and simple as possible. It should do only what has to be done immediately after the event which caused it: operations which can be deferred should be delegated to the main program loop. Typically an ISR will deal with the hardware device which caused the interrupt, making it ready for the next interrupt to occur. It will communicate with the main loop by updating shared data to indicate that the interrupt has occurred, and it will return. An ISR should return control to the main loop as quickly as possible. This is not a specific MicroPython issue so is covered in more detail [below](https://docs.micropython.org/en/latest/reference/isr_rules.html#isr).

### Communication between an ISR and the main program

Normally an ISR needs to communicate with the main program. The simplest means of doing this is via one or more shared data objects, either declared as global or shared via a class (see below). There are various restrictions and hazards around doing this, which are covered in more detail below. Integers, bytes and bytearray objects are commonly used for this purpose along with arrays (from the array module) which can store various data types.

### The use of object methods as callbacks

MicroPython supports this powerful technique which enables an ISR to share instance variables with the underlying code. It also enables a class implementing a device driver to support multiple device instances. The following example causes two LED’s to flash at different rates.

**import** **pyb**, **micropython**

micropython.alloc\_emergency\_exception\_buf(100)

**class** **Foo**(object):

**def** \_\_init\_\_(self, timer, led):

self.led = led

timer.callback(self.cb)

**def** cb(self, tim):

self.led.toggle()

red = Foo(pyb.Timer(4, freq=1), pyb.LED(1))

green = Foo(pyb.Timer(2, freq=0.8), pyb.LED(2))

In this example the red instance associates timer 4 with LED 1: when a timer 4 interrupt occurs red.cb() is called causing LED 1 to change state. The green instance operates similarly: a timer 2 interrupt results in the execution of green.cb() and toggles LED 2. The use of instance methods confers two benefits. Firstly a single class enables code to be shared between multiple hardware instances. Secondly, as a bound method the callback function’s first argument is self. This enables the callback to access instance data and to save state between successive calls. For example,

if the class above had a variable self.count set to zero in the constructor, cb() could increment the counter. The red and green instances would then maintain independent counts of the number of times each LED had changed state.

### Creation of Python objects

ISR’s cannot create instances of Python objects. This is because MicroPython needs to allocate memory for the object from a store of free memory block called the heap. This is not permitted in an interrupt handler because heap allocation is not re-entrant. In other words the interrupt might occur when the main program is part way through performing an allocation - to maintain the integrity of the heap the interpreter disallows memory allocations in ISR code.

A consequence of this is that ISR’s can’t use floating point arithmetic; this is because floats are Python objects. Similarly an ISR can’t append an item to a list. In practice it can be hard to determine exactly which code constructs will attempt to perform memory allocation and provoke an error message: another reason for keeping ISR code short and simple.

One way to avoid this issue is for the ISR to use pre-allocated buffers. For example a class constructor creates a bytearray instance and a boolean flag. The ISR method assigns data to locations in the buffer and sets the flag. The memory allocation occurs in the main program code when the object is instantiated rather than in the ISR.

The MicroPython library I/O methods usually provide an option to use a pre-allocated buffer.

For example pyb.i2c.recv() can accept a mutable buffer as its first argument: this enables its use in an ISR.

A means of creating an object without employing a class or globals is as follows:

**def** set\_volume(t, buf=bytearray(3)):

buf[0] = 0xa5

buf[1] = t >> 4

buf[2] = 0x5a

**return** buf

The compiler instantiates the default buf argument when the function is loaded for the first time (usually when the module it’s in is imported).

An instance of object creation occurs when a reference to a bound method is created. This means that an ISR cannot pass a bound method to a function. One solution is to create a reference to the bound method in the class constructor and to pass that reference in the ISR. For example:

**class** **Foo**():

**def** \_\_init\_\_(self):

self.bar\_ref = self.bar *# Allocation occurs here*

self.x = 0.1

tim = pyb.Timer(4)

tim.init(freq=2)

tim.callback(self.cb)

**def** bar(self, \_):

self.x \*= 1.2

print(self.x)

**def** cb(self, t):

*# Passing self.bar would cause allocation.*

micropython.schedule(self.bar\_ref, 0)

Other techniques are to define and instantiate the method in the constructor or to pass **Foo.bar()** with the argument self.

### Use of Python objects

A further restriction on objects arises because of the way Python works. When an import statement is executed the Python code is compiled to bytecode, with one line of code typically mapping to multiple bytecodes. When the code runs the interpreter reads each bytecode and executes it as a series of machine code instructions. Given that an interrupt can occur at any time between machine code instructions, the original line of Python code may be only partially executed. Consequently a Python object such as a set, list or dictionary modified in the main loop may lack internal consistency at the moment the interrupt occurs.

A typical outcome is as follows. On rare occasions the ISR will run at the precise moment in time when the object is partially updated. When the ISR tries to read the object, a crash results. Because such problems typically occur on rare, random occasions they can be hard to diagnose. There are ways to circumvent this issue, described in [Critical Sections](https://docs.micropython.org/en/latest/reference/isr_rules.html#critical) below.

It is important to be clear about what constitutes the modification of an object. An alteration to a built-in type such as a dictionary is problematic. Altering the contents of an array or bytearray is not. This is because bytes or words are written as a single machine code instruction which is not interruptible: in the parlance of real time programming the write is atomic. A user defined object might instantiate an integer, array or bytearray. It is valid for both the main loop and the ISR to alter the contents of these.

MicroPython supports integers of arbitrary precision. Values between 2\*\*30 -1 and -2\*\*30 will be stored in a single machine word. Larger values are stored as Python objects. Consequently changes to long integers cannot be considered atomic. The use of long integers in ISR’s is unsafe because memory allocation may be attempted as the variable’s value changes.

### Overcoming the float limitation

In general it is best to avoid using floats in ISR code: hardware devices normally handle integers and conversion to floats is normally done in the main loop. However there are a few DSP algorithms which require floating point. On platforms with hardware floating point (such as the Pyboard) the inline ARM Thumb assembler can be used to work round this limitation. This is because the processor stores float values in a machine word; values can be shared between the ISR and main program code via an array of floats.

### Using micropython.schedule

This function enables an ISR to schedule a callback for execution “very soon”. The callback is queued for execution which will take place at a time when the heap is not locked. Hence it can create Python objects and use floats. The callback is also guaranteed to run at a time when the main program has completed any update of Python objects, so the callback will not encounter partially updated objects.

Typical usage is to handle sensor hardware. The ISR acquires data from the hardware and enables it to issue a further interrupt. It then schedules a callback to process the data.

Scheduled callbacks should comply with the principles of interrupt handler design outlined below. This is to avoid problems resulting from I/O activity and the modification of shared data which can arise in any code which pre-empts the main program loop.

Execution time needs to be considered in relation to the frequency with which interrupts can occur. If an interrupt occurs while the previous callback is executing, a further instance of the callback will be queued for execution; this will run after the current instance has completed. A sustained high interrupt repetition rate therefore carries a risk of unconstrained queue growth and eventual failure with a RuntimeError.

If the callback to be passed to [**schedule()**](https://docs.micropython.org/en/latest/library/micropython.html#micropython.schedule) is a bound method, consider the note in “Creation of Python objects”.

## Exceptions

If an ISR raises an exception it will not propagate to the main loop. The interrupt will be disabled unless the exception is handled by the ISR code.

## General issues

This is merely a brief introduction to the subject of real time programming. Beginners should note that design errors in real time programs can lead to faults which are particularly hard to diagnose. This is because they can occur rarely and at intervals which are essentially random. It is crucial to get the initial design right and to anticipate issues before they arise. Both interrupt handlers and the main program need to be designed with an appreciation of the following issues.

### Interrupt handler design

As mentioned above, ISR’s should be designed to be as simple as possible. They should always return in a short, predictable period of time. This is important because when the ISR is running, the main loop is not: inevitably the main loop experiences pauses in its execution at random points in the code. Such pauses can be a source of hard to diagnose bugs particularly if their duration is long or variable. In order to understand the implications of ISR run time, a basic grasp of interrupt priorities is required.

Interrupts are organised according to a priority scheme. ISR code may itself be interrupted by a higher priority interrupt. This has implications if the two interrupts share data (see Critical Sections below). If such an interrupt occurs it interposes a delay into the ISR code. If a lower priority interrupt occurs while the ISR is running, it will be delayed until the ISR is complete: if the delay is too long, the lower priority interrupt may fail. A further issue with slow ISR’s is the case where a second interrupt of the same type occurs during its execution. The second interrupt will be handled on termination of the first. However if the rate of incoming interrupts consistently exceeds the capacity of the ISR to service them the outcome will not be a happy one.

Consequently looping constructs should be avoided or minimised. I/O to devices other than to the interrupting device should normally be avoided: I/O such as disk access, print statements and UART access is relatively slow, and its duration may vary. A further issue here is that filesystem functions are not reentrant: using filesystem I/O in an ISR and the main program would be hazardous. Crucially ISR code should not wait on an event. I/O is acceptable if the code can be guaranteed to return in a predictable period, for example toggling a pin or LED. Accessing the interrupting device via I2C or SPI may be necessary but the time taken for such accesses should be calculated or measured and its impact on the application assessed.

There is usually a need to share data between the ISR and the main loop. This may be done either through global variables or via class or instance variables. Variables are typically integer or boolean types, or integer or byte arrays (a pre-allocated integer array offers faster access than a list). Where multiple values are modified by the ISR it is necessary to consider the case where the interrupt occurs at a time when the main program has accessed some, but not all, of the values. This can lead to inconsistencies.

Consider the following design. An ISR stores incoming data in a bytearray, then adds the number of bytes received to an integer representing total bytes ready for processing. The main program reads the number of bytes, processes the bytes, then clears down the number of bytes ready. This will work until an interrupt occurs just after the main program has read the number of bytes. The ISR puts the added data into the buffer and updates the number received, but the main program has already read the number, so processes the data originally received. The newly arrived bytes are lost.

There are various ways of avoiding this hazard, the simplest being to use a circular buffer. If it is not possible to use a structure with inherent thread safety other ways are described below.

### Reentrancy

A potential hazard may occur if a function or method is shared between the main program and one or more ISR’s or between multiple ISR’s. The issue here is that the function may itself be interrupted and a further instance of that function run. If this is to occur, the function must be designed to be reentrant. How this is done is an advanced topic beyond the scope of this tutorial.

### Critical sections

An example of a critical section of code is one which accesses more than one variable which can be affected by an ISR. If the interrupt happens to occur between accesses to the individual variables, their values will be inconsistent. This is an instance of a hazard known as a race condition: the ISR and the main program loop race to alter the variables. To avoid inconsistency a means must be employed to ensure that the ISR does not alter the values for the duration of the critical section. One way to achieve this is to issue pyb.disable\_irq() before the start of the section, and pyb.enable\_irq() at the end. Here is an example of this approach:

**import** **pyb**, **micropython**, **array**

micropython.alloc\_emergency\_exception\_buf(100)

**class** **BoundsException**(Exception):

**pass**

ARRAYSIZE = const(20)

index = 0

data = array.array('i', 0 **for** x **in** range(ARRAYSIZE))

**def** callback1(t):

**global** data, index

**for** x **in** range(5):

data[index] = pyb.rng() *# simulate input*

index += 1

**if** index >= ARRAYSIZE:

**raise** BoundsException('Array bounds exceeded')

tim4 = pyb.Timer(4, freq=100, callback=callback1)

**for** loop **in** range(1000):

**if** index > 0:

irq\_state = pyb.disable\_irq() *# Start of critical section*

**for** x **in** range(index):

print(data[x])

index = 0

pyb.enable\_irq(irq\_state) *# End of critical section*

print('loop *{}*'.format(loop))

pyb.delay(1)

tim4.callback(**None**)

A critical section can comprise a single line of code and a single variable. Consider the following code fragment.

count = 0

**def** cb(): *# An interrupt callback*

count +=1

**def** main():

*# Code to set up the interrupt callback omitted*

**while** **True**:

count += 1

This example illustrates a subtle source of bugs. The line count += 1 in the main loop carries a specific race condition hazard known as a read-modify-write. This is a classic cause of bugs in real time systems. In the main loop MicroPython reads the value of t.counter, adds 1 to it, and writes it back. On rare occasions the interrupt occurs after the read and before the write. The interrupt modifies t.counter but its change is overwritten by the main loop when the ISR returns. In a real system this could lead to rare, unpredictable failures.

As mentioned above, care should be taken if an instance of a Python built in type is modified in the main code and that instance is accessed in an ISR. The code performing the modification should be regarded as a critical section to ensure that the instance is in a valid state when the ISR runs.

Particular care needs to be taken if a dataset is shared between different ISR’s. The hazard here is that the higher priority interrupt may occur when the lower priority one has partially updated the shared data. Dealing with this situation is an advanced topic beyond the scope of this introduction other than to note that mutex objects described below can sometimes be used.

Disabling interrupts for the duration of a critical section is the usual and simplest way to proceed, but it disables all interrupts rather than merely the one with the potential to cause problems. It is generally undesirable to disable an interrupt for long. In the case of timer interrupts it introduces variability to the time when a callback occurs. In the case of device interrupts, it can lead to the device being serviced too late with possible loss of data or overrun errors in the device hardware. Like ISR’s, a critical section in the main code should have a short, predictable duration.

An approach to dealing with critical sections which radically reduces the time for which interrupts are disabled is to use an object termed a mutex (name derived from the notion of mutual exclusion). The main program locks the mutex before running the critical section and unlocks it at the end. The ISR tests whether the mutex is locked. If it is, it avoids the critical section and returns. The design challenge is defining what the ISR should do in the event that access to the critical variables is denied. A simple example of a mutex may be found [here](https://github.com/peterhinch/micropython-samples.git). Note that the mutex code does disable interrupts, but only for the duration of eight machine instructions: the benefit of this approach is that other interrupts are virtually unaffected.

### Interrupts and the REPL

Interrupt handlers, such as those associated with timers, can continue to run after a program terminates. This may produce unexpected results where you might have expected the object raising the callback to have gone out of scope. For example on the Pyboard:

**def** bar():

foo = pyb.Timer(2, freq=4, callback=**lambda** t: print('.', end=''))

bar()

This continues to run until the timer is explicitly disabled or the board is reset with ctrl D.

# Maximizing MicroPython speed

This tutorial describes ways of improving the performance of MicroPython code. Optimisations involving other languages are covered elsewhere, namely the use of modules written in C and the MicroPython inline assembler.

## Designing for speed

Performance issues should be considered at the outset. This involves taking a view on the sections of code which are most performance critical and devoting particular attention to their design. The process of optimisation begins when the code has been tested: if the design is correct at the outset optimisation will be straightforward and may actually be unnecessary.

## [Algorithms](https://docs.micropython.org/en/latest/reference/speed_python.html#id3)

The most important aspect of designing any routine for performance is ensuring that the best algorithm is employed. This is a topic for textbooks rather than for a MicroPython guide but spectacular performance gains can sometimes be achieved by adopting algorithms known for their efficiency.

## [RAM allocation](https://docs.micropython.org/en/latest/reference/speed_python.html#id4)

To design efficient MicroPython code it is necessary to have an understanding of the way the interpreter allocates RAM. When an object is created or grows in size (for example where an item is appended to a list) the necessary RAM is allocated from a block known as the heap. This takes a significant amount of time; further it will on occasion trigger a process known as garbage collection which can take several milliseconds.

Consequently the performance of a function or method can be improved if an object is created once only and not permitted to grow in size. This implies that the object persists for the duration of its use: typically it will be instantiated in a class constructor and used in various methods.

This is covered in further detail [Controlling garbage collection](https://docs.micropython.org/en/latest/reference/speed_python.html#controlling-gc) below.

## [Buffers](https://docs.micropython.org/en/latest/reference/speed_python.html#id5)

An example of the above is the common case where a buffer is required, such as one used for communication with a device. A typical driver will create the buffer in the constructor and use it in its I/O methods which will be called repeatedly.

The MicroPython libraries typically provide support for pre-allocated buffers. For example, objects which support stream interface (e.g., file or UART) provide read() method which allocates new buffer for read data, but also a readinto() method to read data into an existing buffer.

## [Floating point](https://docs.micropython.org/en/latest/reference/speed_python.html#id6)

Some MicroPython ports allocate floating point numbers on heap. Some other ports may lack dedicated floating-point coprocessor, and perform arithmetic operations on them in “software” at considerably lower speed than on integers. Where performance is important, use integer operations and restrict the use of floating point to sections of the code where performance is not paramount. For example, capture ADC readings as integers values to an array in one quick go, and only then convert them to floating-point numbers for signal processing.

## [Arrays](https://docs.micropython.org/en/latest/reference/speed_python.html#id7)

Consider the use of the various types of array classes as an alternative to lists. The [**array**](https://docs.micropython.org/en/latest/library/uarray.html#uarray.array) module supports various element types with 8-bit elements supported by Python’s built in [**bytes**](https://docs.micropython.org/en/latest/library/builtins.html#bytes) and [**bytearray**](https://docs.micropython.org/en/latest/library/builtins.html#bytearray) classes. These data structures all store elements in contiguous memory locations. Once again to avoid memory allocation in critical code these should be pre-allocated and passed as arguments or as bound objects.

When passing slices of objects such as [**bytearray**](https://docs.micropython.org/en/latest/library/builtins.html#bytearray) instances, Python creates a copy which involves allocation of the size proportional to the size of slice. This can be alleviated using a [**memoryview**](https://docs.micropython.org/en/latest/library/builtins.html#memoryview) object. [**memoryview**](https://docs.micropython.org/en/latest/library/builtins.html#memoryview) itself is allocated on heap, but is a small, fixed-size object, regardless of the size of slice it points too.

ba = bytearray(10000) *# big array*

func(ba[30:2000]) *# a copy is passed, ~2K new allocation*

mv = memoryview(ba) *# small object is allocated*

func(mv[30:2000]) *# a pointer to memory is passed*

A [**memoryview**](https://docs.micropython.org/en/latest/library/builtins.html#memoryview) can only be applied to objects supporting the buffer protocol - this includes arrays but not lists. Small caveat is that while memoryview object is live, it also keeps alive the original buffer object. So, a memoryview isn’t a universal panacea. For instance, in the example above, if you are done with 10K buffer and just need those bytes 30:2000 from it, it may be better to make a slice, and let the 10K buffer go (be ready for garbage collection), instead of making a long-living memoryview and keeping 10K blocked for GC.

Nonetheless, [**memoryview**](https://docs.micropython.org/en/latest/library/builtins.html#memoryview) is indispensable for advanced preallocated buffer management. readinto() method discussed above puts data at the beginning of buffer and fills in entire buffer. What if you need to put data in the middle of existing buffer? Just create a memoryview into the needed section of buffer and pass it to readinto().

## [Identifying the slowest section of code](https://docs.micropython.org/en/latest/reference/speed_python.html#id8)

This is a process known as profiling and is covered in textbooks and (for standard Python) supported by various software tools. For the type of smaller embedded application likely to be running on MicroPython platforms the slowest function or method can usually be established by judicious use of the timing ticks group of functions documented in [**utime**](https://docs.micropython.org/en/latest/library/utime.html#module-utime). Code execution time can be measured in ms, us, or CPU cycles.

The following enables any function or method to be timed by adding an @timed\_function decorator:

**def** timed\_function(f, \*args, \*\*kwargs):

myname = str(f).split(' ')[1]

**def** new\_func(\*args, \*\*kwargs):

t = utime.ticks\_us()

result = f(\*args, \*\*kwargs)

delta = utime.ticks\_diff(utime.ticks\_us(), t)

print('Function *{}* Time = *{:6.3f}*ms'.format(myname, delta/1000))

**return** result

**return** new\_func

# MicroPython code improvements

## [The const() declaration](https://docs.micropython.org/en/latest/reference/speed_python.html#id10)

MicroPython provides a const() declaration. This works in a similar way to #define in C in that when the code is compiled to bytecode the compiler substitutes the numeric value for the identifier. This avoids a dictionary lookup at runtime. The argument to const() may be anything which, at compile time, evaluates to an integer e.g. 0x100 or 1 << 8.

## [Caching object references](https://docs.micropython.org/en/latest/reference/speed_python.html#id11)

Where a function or method repeatedly accesses objects performance is improved by caching the object in a local variable:

**class** **foo**(object):

**def** \_\_init\_\_(self):

self.ba = bytearray(100)

**def** bar(self, obj\_display):

ba\_ref = self.ba

fb = obj\_display.framebuffer

*# iterative code using these two objects*

This avoids the need repeatedly to look up self.ba and obj\_display.framebuffer in the body of the method bar().

## [Controlling garbage collection](https://docs.micropython.org/en/latest/reference/speed_python.html#id12)

When memory allocation is required, MicroPython attempts to locate an adequately sized block on the heap. This may fail, usually because the heap is cluttered with objects which are no longer referenced by code. If a failure occurs, the process known as garbage collection reclaims the memory used by these redundant objects and the allocation is then tried again - a process which can take several milliseconds.

There may be benefits in pre-empting this by periodically issuing [**gc.collect()**](https://docs.micropython.org/en/latest/library/gc.html#gc.collect). Firstly doing a collection before it is actually required is quicker - typically on the order of 1ms if done frequently. Secondly you can determine the point in code where this time is used rather than have a longer delay occur at random points, possibly in a speed critical section. Finally performing collections regularly can reduce fragmentation in the heap. Severe fragmentation can lead to non-recoverable allocation failures.

## The Native code emitter

This causes the MicroPython compiler to emit native CPU opcodes rather than bytecode. It covers the bulk of the MicroPython functionality, so most functions will require no adaptation (but see below). It is invoked by means of a function decorator:

**@micropython**.native

**def** foo(self, arg):

buf = self.linebuf *# Cached object*

*# code*

There are certain limitations in the current implementation of the native code emitter.

* Context managers are not supported (the with statement).
* Generators are not supported.
* If raise is used an argument must be supplied.

The trade-off for the improved performance (roughly twice as fast as bytecode) is an increase in compiled code size.

## The Viper code emitter

The optimisations discussed above involve standards-compliant Python code. The Viper code emitter is not fully compliant. It supports special Viper native data types in pursuit of performance. Integer processing is non-compliant because it uses machine words: arithmetic on 32 bit hardware is performed modulo 2\*\*32.

Like the Native emitter Viper produces machine instructions but further optimisations are performed, substantially increasing performance especially for integer arithmetic and bit manipulations. It is invoked using a decorator:

**@micropython**.viper

**def** foo(self, arg: int) -> int:

*# code*

As the above fragment illustrates it is beneficial to use Python type hints to assist the Viper optimiser. Type hints provide information on the data types of arguments and of the return value; these are a standard Python language feature formally defined here [PEP0484](https://www.python.org/dev/peps/pep-0484/). Viper supports its own set of types namely int, uint (unsigned integer), ptr, ptr8, ptr16 and

ptr32. The ptrX types are discussed below. Currently the uint type serves a single purpose: as a type hint for a function return value. If such a function returns 0xffffffff Python will interpret the result as 2\*\*32 -1 rather than as -1.

In addition to the restrictions imposed by the native emitter the following constraints apply:

* Functions may have up to four arguments.
* Default argument values are not permitted.
* Floating point may be used but is not optimised.

Viper provides pointer types to assist the optimiser. These comprise

* ptr Pointer to an object.
* ptr8 Points to a byte.
* ptr16 Points to a 16 bit half-word.
* ptr32 Points to a 32 bit machine word.

The concept of a pointer may be unfamiliar to Python programmers. It has similarities to a Python [**memoryview**](https://docs.micropython.org/en/latest/library/builtins.html#memoryview) object in that it provides direct access to data stored in memory. Items are accessed using subscript notation, but slices are not supported: a pointer can return a single item only. Its purpose is to provide fast random access to data stored in contiguous memory locations - such as data stored in objects which support the buffer protocol, and memory-mapped peripheral registers in a microcontroller. It should be noted that programming using pointers is hazardous: bounds checking is not performed and the compiler does nothing to prevent buffer overrun errors.

Typical usage is to cache variables:

**@micropython**.viper

**def** foo(self, arg: int) -> int:

buf = ptr8(self.linebuf) *# self.linebuf is a bytearray or bytes object*

**for** x **in** range(20, 30):

bar = buf[x] *# Access a data item through the pointer*

*# code omitted*

In this instance the compiler “knows” that buf is the address of an array of bytes; it can emit code to rapidly compute the address of buf[x] at runtime. Where casts are used to convert objects to Viper native types these should be performed at the start of the function rather than in critical timing loops as the cast operation can take several microseconds. The rules for casting are as follows:

* Casting operators are currently: int, bool, uint, ptr, ptr8, ptr16 and ptr32.
* The result of a cast will be a native Viper variable.
* Arguments to a cast can be a Python object or a native Viper variable.
* If argument is a native Viper variable, then cast is a no-op (i.e. costs nothing at runtime) that just changes the type (e.g. from uint to ptr8) so that you can then store/load using this pointer.
* If the argument is a Python object and the cast is int or uint, then the Python object must be of integral type and the value of that integral object is returned.
* The argument to a bool cast must be integral type (boolean or integer); when used as a return type the viper function will return True or False objects.
* If the argument is a Python object and the cast is ptr, ptr, ptr16 or ptr32, then the Python object must either have the buffer protocol (in which case a pointer to the start of the buffer is returned) or it must be of integral type (in which case the value of that integral object is returned).

Writing to a pointer which points to a read-only object will lead to undefined behaviour.

The following example illustrates the use of a ptr16 cast to toggle pin X1 n times:

BIT0 = const(1)

**@micropython**.viper

**def** toggle\_n(n: int):

odr = ptr16(stm.GPIOA + stm.GPIO\_ODR)

**for** \_ **in** range(n):

odr[0] ^= BIT0

A detailed technical description of the three code emitters may be found on Kickstarter here [Note 1](https://www.kickstarter.com/projects/214379695/micro-python-python-for-microcontrollers/posts/664832) and here [Note 2](https://www.kickstarter.com/projects/214379695/micro-python-python-for-microcontrollers/posts/665145)

## Accessing hardware directly

**Note**

Code examples in this section are given for the Pyboard. The techniques described however may be applied to other MicroPython ports too.

This comes into the category of more advanced programming and involves some knowledge of the target MCU. Consider the example of toggling an output pin on the Pyboard. The standard approach would be to write

mypin.value(mypin.value() ^ 1) *# mypin was instantiated as an output pin*

This involves the overhead of two calls to the [Pin](https://docs.micropython.org/en/latest/library/machine.Pin.html#machine.Pin) instance’s [value()](https://docs.micropython.org/en/latest/library/machine.Pin.html#machine.Pin.value) method. This overhead can be eliminated by performing a read/write to the relevant bit of the chip’s GPIO port output data register (odr). To facilitate this the stm module provides a set of constants providing the addresses of the relevant registers. A fast toggle of pin P4 (CPU pin A14) - corresponding to the green LED - can be performed as follows:

**import** **machine**

**import** **stm**

BIT14 = const(1 << 14)

machine.mem16[stm.GPIOA + stm.GPIO\_ODR] ^= BIT14

# MicroPython on microcontrollers

MicroPython is designed to be capable of running on microcontrollers. These have hardware limitations which may be unfamiliar to programmers more familiar with conventional computers. In particular the amount of RAM and nonvolatile “disk” (flash memory) storage is limited. This tutorial offers ways to make the most of the limited resources. Because MicroPython runs on controllers based on a variety of architectures, the methods presented are generic: in some cases it will be necessary to obtain detailed information from platform specific documentation.

## Flash memory

On the Pyboard the simple way to address the limited capacity is to fit a micro SD card. In some cases this is impractical, either because the device does not have an SD card slot or for reasons of cost or power consumption; hence the on-chip flash must be used. The firmware including the MicroPython subsystem is stored in the onboard flash. The remaining capacity is available for use. For reasons connected with the physical architecture of the flash memory part of this capacity may be inaccessible as a filesystem. In such cases this space may be employed by incorporating user modules into a firmware build which is then flashed to the device.

There are two ways to achieve this: frozen modules and frozen bytecode. Frozen modules store the Python source with the firmware. Frozen bytecode uses the cross compiler to convert the source to bytecode which is then stored with the firmware. In either case the module may be accessed with an import statement:

**import** **mymodule**

The procedure for producing frozen modules and bytecode is platform dependent; instructions for building the firmware can be found in the README files in the relevant part of the source tree.

In general terms the steps are as follows:

* Clone the MicroPython [repository](https://github.com/micropython/micropython).
* Acquire the (platform specific) toolchain to build the firmware.
* Build the cross compiler.
* Place the modules to be frozen in a specified directory (dependent on whether the module is to be frozen as source or as bytecode).
* Build the firmware. A specific command may be required to build frozen code of either type - see the platform documentation.
* Flash the firmware to the device.

## RAM

When reducing RAM usage there are two phases to consider: compilation and execution. In addition to memory consumption, there is also an issue known as heap fragmentation. In general terms it is best to minimise the repeated creation and destruction of objects. The reason for this is covered in the section covering the [heap](https://docs.micropython.org/en/latest/reference/constrained.html#heap).

### Compilation phase

When a module is imported, MicroPython compiles the code to bytecode which is then executed by the MicroPython virtual machine (VM). The bytecode is stored in RAM. The compiler itself requires RAM, but this becomes available for use when the compilation has completed.

If a number of modules have already been imported the situation can arise where there is insufficient RAM to run the compiler. In this case the import statement will produce a memory exception.

If a module instantiates global objects on import it will consume RAM at the time of import, which is then unavailable for the compiler to use on subsequent imports. In general it is best to avoid code which runs on import; a better approach is to have initialisation code which is run by the application after all modules have been imported. This maximises the RAM available to the compiler.

If RAM is still insufficient to compile all modules one solution is to precompile modules. MicroPython has a cross compiler capable of compiling Python modules to bytecode (see the README in the mpy-cross directory). The resulting bytecode file has a .mpy extension; it may be copied to the filesystem and imported in the usual way. Alternatively some or all modules may be implemented as frozen bytecode: on most platforms this saves even more RAM as the bytecode is run directly from flash rather than being stored in RAM.

### Execution phase

There are a number of coding techniques for reducing RAM usage.

**Constants**

MicroPython provides a const keyword which may be used as follows:

**from** **micropython** **import** const

ROWS = const(33)

\_COLS = const(0x10)

a = ROWS

b = \_COLS

In both instances where the constant is assigned to a variable the compiler will avoid coding a lookup to the name of the constant by substituting its literal value. This saves bytecode and hence RAM. However the ROWS value will occupy at least two machine words, one each for the key and value in the globals dictionary. The presence in the dictionary is necessary because another module might import or use it. This RAM can be saved by prepending the name with an underscore as in \_COLS: this symbol is not visible outside the module so will not occupy RAM.

The argument to const() may be anything which, at compile time, evaluates to an integer e.g. 0x100 or 1 << 8. It can even include other const symbols that have already been defined, e.g. 1 << BIT.

**Constant data structures**

Where there is a substantial volume of constant data and the platform supports execution from Flash, RAM may be saved as follows. The data should be located in Python modules and frozen as bytecode. The data must be defined as [**bytes**](https://docs.micropython.org/en/latest/library/builtins.html#bytes) objects. The compiler ‘knows’ that [**bytes**](https://docs.micropython.org/en/latest/library/builtins.html#bytes) objects are immutable and ensures that the objects remain in flash memory rather than being copied to RAM. The [**ustruct**](https://docs.micropython.org/en/latest/library/ustruct.html#module-ustruct) module can assist in converting between [**bytes**](https://docs.micropython.org/en/latest/library/builtins.html#bytes) types and other Python built-in types.

When considering the implications of frozen bytecode, note that in Python strings, floats, bytes, integers and complex numbers are immutable. Accordingly these will be frozen into flash. Thus, in the line

mystring = "The quick brown fox"

the actual string “The quick brown fox” will reside in flash. At runtime a reference to the string is assigned to the variable mystring. The reference occupies a single machine word. In principle a long integer could be used to store constant data:

bar = 0xDEADBEEF0000DEADBEEF

As in the string example, at runtime a reference to the arbitrarily large integer is assigned to the variable bar. That reference occupies a single machine word.

It might be expected that tuples of integers could be employed for the purpose of storing constant data with minimal RAM use. With the current compiler this is ineffective (the code works, but RAM is not saved).

foo = (1, 2, 3, 4, 5, 6, 100000)

At runtime the tuple will be located in RAM. This may be subject to future improvement.

**Needless object creation**

There are a number of situations where objects may unwittingly be created and destroyed. This can reduce the usability of RAM through fragmentation. The following sections discuss instances of this.

**String concatenation**

Consider the following code fragments which aim to produce constant strings:

var = "foo" + "bar"

var1 = "foo" "bar"

var2 = """**\**

foo**\**

bar"""

Each produces the same outcome, however the first needlessly creates two string objects at runtime, allocates more RAM for concatenation before producing the third. The others perform the concatenation at compile time which is more efficient, reducing fragmentation.

Where strings must be dynamically created before being fed to a stream such as a file it will save RAM if this is done in a piecemeal fashion. Rather than creating a large string object, create a substring and feed it to the stream before dealing with the next.

The best way to create dynamic strings is by means of the string format() method:

var = "Temperature *{:5.2f}* Pressure *{:06d}***\n**".format(temp, press)

**Buffers**

When accessing devices such as instances of UART, I2C and SPI interfaces, using pre-allocated buffers avoids the creation of needless objects. Consider these two loops:

**while** **True**:

var = spi.read(100)

*# process data*

buf = bytearray(100)

**while** **True**:

spi.readinto(buf)

*# process data in buf*

The first creates a buffer on each pass whereas the second re-uses a pre-allocated buffer; this is both faster and more efficient in terms of memory fragmentation.

**Bytes are smaller than ints**

On most platforms an integer consumes four bytes. Consider the two calls to the function foo():

**def** foo(bar):

**for** x **in** bar:

print(x)

foo((1, 2, 0xff))

foo(b'**\1\2\xff**')

In the first call a tuple of integers is created in RAM. The second efficiently creates a [**bytes**](https://docs.micropython.org/en/latest/library/builtins.html#bytes) object consuming the minimum amount of RAM. If the module were frozen as bytecode, the [**bytes**](https://docs.micropython.org/en/latest/library/builtins.html#bytes) object would reside in flash.

**Strings Versus Bytes**

Python3 introduced Unicode support. This introduced a distinction between a string and an array of bytes. MicroPython ensures that Unicode strings take no additional space so long as all characters in the string are ASCII (i.e. have a value < 126). If values in the full 8-bit range are required [**bytes**](https://docs.micropython.org/en/latest/library/builtins.html#bytes) and [**bytearray**](https://docs.micropython.org/en/latest/library/builtins.html#bytearray) objects can be used to ensure that no additional space will be required. Note that most string methods (e.g. [**str.strip()**](https://docs.python.org/3.5/library/stdtypes.html#str.strip)) apply also to [**bytes**](https://docs.micropython.org/en/latest/library/builtins.html#bytes) instances so the process of eliminating Unicode can be painless.

s = 'the quick brown fox' *# A string instance*

b = b'the quick brown fox' *# A bytes instance*

Where it is necessary to convert between strings and bytes the [**str.encode()**](https://docs.python.org/3.5/library/stdtypes.html#str.encode) and the [**bytes.decode()**](https://docs.python.org/3.5/library/stdtypes.html#bytes.decode) methods can be used. Note that both strings and bytes are immutable. Any operation which takes as input such an object and produces another implies at least one RAM allocation to produce the result. In the second line below a new bytes object is allocated. This would also occur if foo were a string.

foo = b' empty whitespace'

foo = foo.lstrip()

**Runtime compiler execution**

The Python funcitons [**eval**](https://docs.micropython.org/en/latest/library/builtins.html#eval) and [**exec**](https://docs.micropython.org/en/latest/library/builtins.html#exec) invoke the compiler at runtime, which requires significant amounts of RAM. Note that the pickle library from [**micropython-lib**](https://docs.micropython.org/en/latest/reference/glossary.html#term-micropython-lib) employs [**exec**](https://docs.micropython.org/en/latest/library/builtins.html#exec). It may be more RAM efficient to use the [**ujson**](https://docs.micropython.org/en/latest/library/ujson.html#module-ujson) library for object serialisation.

**Storing strings in flash**

Python strings are immutable hence have the potential to be stored in read only memory. The compiler can place in flash strings defined in Python code. As with frozen modules it is necessary to have a copy of the source tree on the PC and the toolchain to build the firmware. The procedure will work even if the modules have not been fully debugged, so long as they can be imported and run.

After importing the modules, execute:

micropython.qstr\_info(1)

Then copy and paste all the Q(xxx) lines into a text editor. Check for and remove lines which are obviously invalid. Open the file qstrdefsport.h which will be found in ports/stm32 (or the equivalent directory for the architecture in use). Copy and paste the corrected lines at the end of the file. Save the file, rebuild and flash the firmware. The outcome can be checked by importing the modules and again issuing:

micropython.qstr\_info(1)

The Q(xxx) lines should be gone.

## The heap

When a running program instantiates an object the necessary RAM is allocated from a fixed size pool known as the heap. When the object goes out of scope (in other words becomes inaccessible to code) the redundant object is known as “garbage”. A process known as “garbage collection” (GC) reclaims that memory, returning it to the free heap. This process runs automatically, however it can be invoked directly by issuing [**gc.collect()**](https://docs.micropython.org/en/latest/library/gc.html#gc.collect).

The discourse on this is somewhat involved. For a ‘quick fix’ issue the following periodically:

gc.collect()

gc.threshold(gc.mem\_free() // 4 + gc.mem\_alloc())

### Fragmentation

Say a program creates an object foo, then an object bar. Subsequently foo goes out of scope but bar remains. The RAM used by foo will be reclaimed by GC. However if bar was allocated to a higher address, the RAM reclaimed from foo will only be of use for objects no bigger than foo. In a complex or long running program the heap can become fragmented: despite there being a substantial amount of RAM available, there is insufficient contiguous space to allocate a particular object, and the program fails with a memory error.

The techniques outlined above aim to minimise this. Where large permanent buffers or other objects are required it is best to instantiate these early in the process of program execution before fragmentation can occur. Further improvements may be made by monitoring the state of the heap and by controlling GC; these are outlined below.

### Reporting

A number of library functions are available to report on memory allocation and to control GC. These are to be found in the [**gc**](https://docs.micropython.org/en/latest/library/gc.html#module-gc) and [**micropython**](https://docs.micropython.org/en/latest/library/micropython.html#module-micropython) modules. The following example may be pasted at the REPL (ctrl e to enter paste mode, ctrl d to run it).

**import** **gc**

**import** **micropython**

gc.collect()

micropython.mem\_info()

print('-----------------------------')

print('Initial free: *{}* allocated: *{}*'.format(gc.mem\_free(), gc.mem\_alloc()))

**def** func():

a = bytearray(10000)

gc.collect()

print('Func definition: *{}* allocated: *{}*'.format(gc.mem\_free(), gc.mem\_alloc()))

func()

print('Func run free: *{}* allocated: *{}*'.format(gc.mem\_free(), gc.mem\_alloc()))

gc.collect()

print('Garbage collect free: *{}* allocated: *{}*'.format(gc.mem\_free(), gc.mem\_alloc()))

print('-----------------------------')

micropython.mem\_info(1)

Methods employed above:

* [**gc.collect()**](https://docs.micropython.org/en/latest/library/gc.html#gc.collect) Force a garbage collection. See footnote.
* [**micropython.mem\_info()**](https://docs.micropython.org/en/latest/library/micropython.html#micropython.mem_info) Print a summary of RAM utilisation.
* [**gc.mem\_free()**](https://docs.micropython.org/en/latest/library/gc.html#gc.mem_free) Return the free heap size in bytes.
* [**gc.mem\_alloc()**](https://docs.micropython.org/en/latest/library/gc.html#gc.mem_alloc) Return the number of bytes currently allocated.
* micropython.mem\_info(1) Print a table of heap utilisation (detailed below).

The numbers produced are dependent on the platform, but it can be seen that declaring the function uses a small amount of RAM in the form of bytecode emitted by the compiler (the RAM used by the compiler has been reclaimed). Running the function uses over 10KiB, but on return a is garbage because it is out of scope and cannot be referenced. The final [**gc.collect()**](https://docs.micropython.org/en/latest/library/gc.html#gc.collect) recovers that memory.

The final output produced by micropython.mem\_info(1) will vary in detail but may be interpreted as follows:

| **Symbol** | **Meaning** |
| --- | --- |
| . | free block |
| h | head block |
| = | tail block |
| m | marked head block |
| T | tuple |
| L | list |
| D | dict |
| F | float |
| B | byte code |
| M | module |

Each letter represents a single block of memory, a block being 16 bytes. So each line of the heap dump represents 0x400 bytes or 1KiB of RAM.

### Control of garbage collection

A GC can be demanded at any time by issuing [**gc.collect()**](https://docs.micropython.org/en/latest/library/gc.html#gc.collect). It is advantageous to do this at intervals, firstly to pre-empt fragmentation and secondly for performance. A GC can take several milliseconds but is quicker when there is little work to do (about 1ms on the Pyboard). An explicit call can minimise that delay while ensuring it occurs at points in the program when it is acceptable.

Automatic GC is provoked under the following circumstances. When an attempt at allocation fails, a GC is performed and the allocation re-tried. Only if this fails is an exception raised. Secondly an automatic GC will be triggered if the amount of free RAM falls below a threshold. This threshold can be adapted as execution progresses:

gc.collect()

gc.threshold(gc.mem\_free() // 4 + gc.mem\_alloc())

This will provoke a GC when more than 25% of the currently free heap becomes occupied.

In general modules should instantiate data objects at runtime using constructors or other initialisation functions. The reason is that if this occurs on initialisation the compiler may be starved of RAM when subsequent modules are imported. If modules do instantiate data on import then [**gc.collect()**](https://docs.micropython.org/en/latest/library/gc.html#gc.collect) issued after the import will ameliorate the problem.

## String operations

MicroPython handles strings in an efficient manner and understanding this can help in designing applications to run on microcontrollers. When a module is compiled, strings which occur multiple times are stored once only, a process known as string interning. In MicroPython an interned string is known as a qstr. In a module imported normally that single instance will be located in RAM, but as described above, in modules frozen as bytecode it will be located in flash.

String comparisons are also performed efficiently using hashing rather than character by character. The penalty for using strings rather than integers may hence be small both in terms of performance and RAM usage - a fact which may come as a surprise to C programmers.

## Postscript

MicroPython passes, returns and (by default) copies objects by reference. A reference occupies a single machine word so these processes are efficient in RAM usage and speed.

Where variables are required whose size is neither a byte nor a machine word there are standard libraries which can assist in storing these efficiently and in performing conversions. See the [**array**](https://docs.micropython.org/en/latest/library/uarray.html#uarray.array), [**ustruct**](https://docs.micropython.org/en/latest/library/ustruct.html#module-ustruct) and [**uctypes**](https://docs.micropython.org/en/latest/library/uctypes.html#module-uctypes) modules.

### Footnote: gc.collect() return value

On Unix and Windows platforms the [**gc.collect()**](https://docs.micropython.org/en/latest/library/gc.html#gc.collect) method returns an integer which signifies the number of distinct memory regions that were reclaimed in the collection (more precisely, the number of heads that were turned into frees). For efficiency reasons bare metal ports do not return this value.

# Distribution packages, package management, and deploying applications

Just as the “big” Python, MicroPython supports creation of “third party” packages, distributing them, and easily installing them in each user’s environment. This chapter discusses how these actions are achieved. Some familiarity with Python packaging is recommended.

**Overview**

Steps below represent a high-level workflow when creating and consuming packages:

1. Python modules and packages are turned into distribution package archives, and published at the Python Package Index (PyPI).
2. [**upip**](https://docs.micropython.org/en/latest/reference/glossary.html#term-upip) package manager can be used to install a distribution package on a [**MicroPython port**](https://docs.micropython.org/en/latest/reference/glossary.html#term-micropython-port) with networking capabilities (for example, on the Unix port).
3. For ports without networking capabilities, an “installation image” can be prepared on the Unix port, and transferred to a device by suitable means.
4. For low-memory ports, the installation image can be frozen as the bytecode into MicroPython executable, thus minimizing the memory storage overheads.

The sections below describe this process in details.

**Distribution packages**

Python modules and packages can be packaged into archives suitable for transfer between systems, storing at the well-known location (PyPI), and downloading on demand for deployment. These archives are known as *distribution packages* (to differentiate them from Python packages (means to organize Python source code)).

The MicroPython distribution package format is a well-known tar.gz format, with some adaptations however. The Gzip compressor, used as an external wrapper for TAR archives, by default uses 32KB dictionary size, which means that to uncompress a compressed stream, 32KB of contiguous memory needs to be allocated. This requirement may be not satisfiable on low-memory devices, which may have total memory available less than that amount, and even if not, a contiguous block like that may be hard to allocate due to memory fragmentation. To accommodate these constraints, MicroPython distribution packages use Gzip compression with the dictionary size of 4K, which should be a suitable compromise with still achieving some compression while being able to uncompressed even by the smallest devices.

Besides the small compression dictionary size, MicroPython distribution packages also have other optimizations, like removing any files from the archive which aren’t used by the installation process. In particular, [**upip**](https://docs.micropython.org/en/latest/reference/glossary.html#term-upip) package manager doesn’t execute setup.py during installation (see below), and thus that file is not included in the archive.

At the same time, these optimizations make MicroPython distribution packages not compatible with [**CPython**](https://docs.micropython.org/en/latest/reference/glossary.html#term-cpython)‘s package manager, pip. This isn’t considered a big problem, because:

1. Packages can be installed with [**upip**](https://docs.micropython.org/en/latest/reference/glossary.html#term-upip), and then can be used with CPython (if they are compatible with it).
2. In the other direction, majority of CPython packages would be incompatible with MicroPython by various reasons, first of all, the reliance on features not implemented by MicroPython.

Summing up, the MicroPython distribution package archives are highly optimized for MicroPython’s target environments, which are highly resource constrained devices.

**upip package manager**

MicroPython distribution packages are intended to be installed using the [**upip**](https://docs.micropython.org/en/latest/reference/glossary.html#term-upip) package manager. [**upip**](https://docs.micropython.org/en/latest/reference/glossary.html#term-upip) is a Python application which is usually distributed (as frozen bytecode) with network-enabled [**MicroPython ports**](https://docs.micropython.org/en/latest/reference/glossary.html#term-micropython-port). At the very least, [**upip**](https://docs.micropython.org/en/latest/reference/glossary.html#term-upip) is available in the [**MicroPython Unix port**](https://docs.micropython.org/en/latest/reference/glossary.html#term-micropython-unix-port).

On any [**MicroPython port**](https://docs.micropython.org/en/latest/reference/glossary.html#term-micropython-port) providing [**upip**](https://docs.micropython.org/en/latest/reference/glossary.html#term-upip), it can be accessed as following:

**import** **upip**

upip.help()

upip.install(package\_or\_package\_list, [path])

Where *package\_or\_package\_list* is the name of a distribution package to install, or a list of such names to install multiple packages. Optional *path* parameter specifies filesystem location to install under and defaults to the standard library location (see below).

An example of installing a specific package and then using it:

**>>> import** **upip**

**>>>** upip.install("micropython-pystone\_lowmem")

[...]

**>>> import** **pystone\_lowmem**

**>>>** pystone\_lowmem.main()

Note that the name of Python package and the name of distribution package for it in general don’t have to match, and oftentimes they don’t. This is because PyPI provides a central package repository for all different Python implementations and versions, and thus distribution package names may need to be namespaced for a particular implementation. For example, all packages from [**micropython-lib**](https://docs.micropython.org/en/latest/reference/glossary.html#term-micropython-lib) follow this naming convention: for a Python module or package named foo, the distribution package name is micropython-foo.

For the ports which run MicroPython executable from the OS command prompts (like the Unix port), [**upip**](https://docs.micropython.org/en/latest/reference/glossary.html#term-upip) can be (and indeed, usually is) run from the command line instead of MicroPython’s own REPL. The commands which corresponds to the example above are:

micropython -m upip -h

micropython -m upip install [-p <path>] <packages>...

micropython -m upip install micropython-pystone\_lowmem

[TODO: Describe installation path.]

**Cross-installing packages**

For [**MicroPython ports**](https://docs.micropython.org/en/latest/reference/glossary.html#term-micropython-port) without native networking capabilities, the recommend process is “cross-installing” them into a “directory image” using the [**MicroPython Unix port**](https://docs.micropython.org/en/latest/reference/glossary.html#term-micropython-unix-port), and then transferring this image to a device by suitable means.

Installing to a directory image involves using -p switch to [**upip**](https://docs.micropython.org/en/latest/reference/glossary.html#term-upip):

micropython -m upip install -p install\_dir micropython-pystone\_lowmem

After this command, the package content (and contents of every dependency packages) will be available in the install\_dir/ subdirectory. You would need to transfer contents of this directory (without the install\_dir/ prefix) to the device, at the suitable location, where it can be found by the Python import statement (see discussion of the [**upip**](https://docs.micropython.org/en/latest/reference/glossary.html#term-upip) installation path above).

**Cross-installing packages with freezing**

For the low-memory [**MicroPython ports**](https://docs.micropython.org/en/latest/reference/glossary.html#term-micropython-port), the process described in the previous section does not provide the most efficient resource usage,because the packages are installed in the source form, so need to be compiled to the bytecome on each import. This compilation requires RAM, and the resulting bytecode is also stored in RAM, reducing its amount available for storing application data. Moreover, the process above requires presence of the filesystem on a device, and the most resource-constrained devices may not even have it.

The bytecode freezing is a process which resolves all the issues mentioned above:

* The source code is pre-compiled into bytecode and store as such.
* The bytecode is stored in ROM, not RAM.
* Filesystem is not required for frozen packages.

Using frozen bytecode requires building the executable (firmware) for a given [**MicroPython port**](https://docs.micropython.org/en/latest/reference/glossary.html#term-micropython-port) from the C source code. Consequently, the process is:

1. Follow the instructions for a particular port on setting up a toolchain and building the port. For example, for ESP8266 port, study instructions in ports/esp8266/README.md and follow them. Make sure you can build the port and deploy the resulting executable/firmware successfully before proceeding to the next steps.
2. Build [**MicroPython Unix port**](https://docs.micropython.org/en/latest/reference/glossary.html#term-micropython-unix-port) and make sure it is in your PATH and you can execute micropython.
3. Change to port’s directory (e.g. ports/esp8266/ for ESP8266).
4. Run make clean-frozen. This step cleans up any previous modules which were installed for freezing (consequently, you need to skip this step to add additional modules, instead of starting from scratch).
5. Run micropython -m upip install -p modules <packages>... to install packages you want to freeze.
6. Run make clean.
7. Run make.

After this, you should have the executable/firmware with modules as the bytecode inside, which you can deploy the usual way.

Few notes:

1. Step 5 in the sequence above assumes that the distribution package is available from PyPI. If that is not the case, you would need to copy Python source files manually to modules/ subdirectory of the port port directory. (Note that upip does not support installing from e.g. version control repositories).
2. The firmware for baremetal devices usually has size restrictions, so adding too many frozen modules may overflow it. Usually, you would get a linking error if this happens. However, in some cases, an image may be produced, which is not runnable on a device. Such cases are in general bugs, and should be reported and further investigated. If you face such a situation, as an initial step, you may want to decrease the amount of frozen modules included.

**Creating distribution packages**

Distribution packages for MicroPython are created in the same manner as for CPython or any other Python implementation, see references at the end of chapter. Setuptools (instead of distutils) should be used, because distutils do not support dependencies and other features. “Source distribution” (sdist) format is used for packaging. The post-processing discussed above, (and pre-processing discussed in the following section) is achieved by using custom sdist command for setuptools. Thus, packaging steps remain the same as for the standard setuptools, the user just needs to override sdist command implementation by passing the appropriate argument to setup() call:

**from** **setuptools** **import** setup

**import** **sdist\_upip**

setup(

...,

cmdclass={'sdist': sdist\_upip.sdist}

)

The sdist\_upip.py module as referenced above can be found in [**micropython-lib**](https://docs.micropython.org/en/latest/reference/glossary.html#term-micropython-lib):

<https://github.com/micropython/micropython-lib/blob/master/sdist_upip.py>

**Application resources**

A complete application, besides the source code, oftentimes also consists of data files, e.g. web page templates, game images, etc. It’s clear how to deal with those when application is installed manually - you just put those data files in the filesystem at some location and use the normal file access functions.

The situation is different when deploying applications from packages - this is more advanced, streamlined and flexible way, but also requires more advanced approach to accessing data files. This approach is treating the data files as “resources”, and abstracting away access to them.

Python supports resource access using its “setuptools” library, using pkg\_resources module. MicroPython, following its usual approach, implements subset of the functionality of that module, specifically pkg\_resources.resource\_stream(package, resource) function. The idea is that an application calls this function, passing a resource identifier, which is a relative path to data file within the specified package (usually top-level application package). It returns a stream object which can be used to access resource contents. Thus, the resource\_stream() emulates interface of the standard [**open()**](https://docs.micropython.org/en/latest/library/builtins.html#open) function.

Implementation-wise, resource\_stream() uses file operations underlyingly, if distribution package is install in the filesystem. However, it also supports functioning without the underlying filesystem, e.g. if the package is frozen as the bytecode. This however requires an extra intermediate step when packaging application - creation of “Python resource module”.

The idea of this module is to convert binary data to a Python bytes object, and put it into the dictionary, indexed by the resource name. This conversion is done automatically using overridden sdist command described in the previous section.

Let’s trace the complete process using the following example. Suppose your application has the following structure:

my\_app/

\_\_main\_\_.py

utils.py

data/

page.html

image.png

\_\_main\_\_.py and utils.py should access resources using the following calls:

**import** **pkg\_resources**

pkg\_resources.resource\_stream(\_\_name\_\_, "data/page.html")

pkg\_resources.resource\_stream(\_\_name\_\_, "data/image.png")

You can develop and debug using the [**MicroPython Unix port**](https://docs.micropython.org/en/latest/reference/glossary.html#term-micropython-unix-port) as usual. When time comes to make a distribution package out of it, just use overridden “sdist” command from sdist\_upip.py module as described in the previous section.

This will create a Python resource module named R.py, based on the files declared in MANIFEST or MANIFEST.in files (any non-.py file will be considered a resource and added to R.py) - before proceeding with the normal packaging steps.

Prepared like this, your application will work both when deployed to filesystem and as frozen bytecode.

If you would like to debug R.py creation, you can run:

python3 setup.py sdist --manifest-only

Alternatively, you can use tools/mpy\_bin2res.py script from the MicroPython distribution, in which can you will need to pass paths to all resource files:

mpy\_bin2res.py data/page.html data/image.png

**References**

Python Packaging User Guide: <https://packaging.python.org/>

Setuptools documentation: <https://setuptools.readthedocs.io/>

Distutils documentation: <https://docs.python.org/3/library/distutils.html>

# Inline assembler for Thumb2 architectures

This document assumes some familiarity with assembly language programming and should be read after studying the [tutorial](https://docs.micropython.org/en/latest/pyboard/tutorial/assembler.html#pyboard-tutorial-assembler). For a detailed description of the instruction set consult the Architecture Reference Manual detailed below. The inline assembler supports a subset of the ARM Thumb-2 instruction set described here. The syntax tries to be as close as possible to that defined in the above ARM manual, converted to Python function calls.

Instructions operate on 32 bit signed integer data except where stated otherwise. Most supported instructions operate on registers R0-R7 only: where R8-R15 are supported this is stated. Registers R8-R12 must be restored to their initial value before return from a function. Registers R13-R15 constitute the Link Register, Stack Pointer and Program Counter respectively.

**Document conventions**

Where possible the behaviour of each instruction is described in Python, for example

* add(Rd, Rn, Rm) Rd = Rn + Rm

This enables the effect of instructions to be demonstrated in Python. In certain case this is impossible because Python doesn’t support concepts such as indirection. The pseudocode employed in such cases is described on the relevant page.

**Instruction categories**

The following sections details the subset of the ARM Thumb-2 instruction set supported by MicroPython.

1. [Register move instructions](https://docs.micropython.org/en/latest/reference/asm_thumb2_mov.html)
2. [Load register from memory](https://docs.micropython.org/en/latest/reference/asm_thumb2_ldr.html)
3. [Store register to memory](https://docs.micropython.org/en/latest/reference/asm_thumb2_str.html)
4. [Logical & bitwise instructions](https://docs.micropython.org/en/latest/reference/asm_thumb2_logical_bit.html)
5. [Arithmetic instructions](https://docs.micropython.org/en/latest/reference/asm_thumb2_arith.html)
6. [Comparison instructions](https://docs.micropython.org/en/latest/reference/asm_thumb2_compare.html)
7. [Branch instructions](https://docs.micropython.org/en/latest/reference/asm_thumb2_label_branch.html)
8. [Stack push and pop](https://docs.micropython.org/en/latest/reference/asm_thumb2_stack.html)
9. [Miscellaneous instructions](https://docs.micropython.org/en/latest/reference/asm_thumb2_misc.html)
10. [Floating point instructions](https://docs.micropython.org/en/latest/reference/asm_thumb2_float.html)
11. [Assembler directives](https://docs.micropython.org/en/latest/reference/asm_thumb2_directives.html)

**Usage examples**

These sections provide further code examples and hints on the use of the assembler.

1. [Hints and tips](https://docs.micropython.org/en/latest/reference/asm_thumb2_hints_tips.html)

**References**

1. [Assembler Tutorial](https://docs.micropython.org/en/latest/pyboard/tutorial/assembler.html#pyboard-tutorial-assembler)
2. [Wiki hints and tips](http://wiki.micropython.org/platforms/boards/pyboard/assembler)
3. [uPy Inline Assembler source-code, emitinlinethumb.c](https://github.com/micropython/micropython/blob/master/py/emitinlinethumb.c)
4. [ARM Thumb2 Instruction Set Quick Reference Card](http://infocenter.arm.com/help/topic/com.arm.doc.qrc0001l/QRC0001_UAL.pdf)
5. [RM0090 Reference Manual](http://www.google.ae/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&cad=rja&uact=8&sqi=2&ved=0CBoQFjAA&url=http%3A%2F%2Fwww.st.com%2Fst-web-ui%2Fstatic%2Factive%2Fen%2Fresource%2Ftechnical%2Fdocument%2Freference_manual%2FDM00031020.pdf&ei=G0rSU66xFeuW0QWYwoD4CQ&usg=AFQjCNFuW6TgzE4QpahO_U7g3f3wdwecAg&sig2=iET-R0y9on_Pbflzf9aYDw&bvm=bv.71778758,bs.1,d.bGQ)

ARM v7-M Architecture Reference Manual (Available on the ARM site after a simple registration procedure. Also available on academic sites but beware of out of date versions.)

# Working with file systems

**Contents**

* [Working with filesystems](https://docs.micropython.org/en/latest/reference/filesystem.html#working-with-filesystems)
* [VFS](https://docs.micropython.org/en/latest/reference/filesystem.html#vfs)
* [Block devices](https://docs.micropython.org/en/latest/reference/filesystem.html#block-devices)
  + [Built-in block devices](https://docs.micropython.org/en/latest/reference/filesystem.html#built-in-block-devices)
    - [STM32 / Pyboard](https://docs.micropython.org/en/latest/reference/filesystem.html#stm32-pyboard)
    - [ESP8266](https://docs.micropython.org/en/latest/reference/filesystem.html#esp8266)
    - [ESP32](https://docs.micropython.org/en/latest/reference/filesystem.html#esp32)
  + [Custom block devices](https://docs.micropython.org/en/latest/reference/filesystem.html#custom-block-devices)
* [Filesystems](https://docs.micropython.org/en/latest/reference/filesystem.html#filesystems)
* [FAT](https://docs.micropython.org/en/latest/reference/filesystem.html#fat)
* [Littlefs](https://docs.micropython.org/en/latest/reference/filesystem.html#littlefs)
* [Hybrid (STM32)](https://docs.micropython.org/en/latest/reference/filesystem.html#hybrid-stm32)
* [Hybrid (ESP32)](https://docs.micropython.org/en/latest/reference/filesystem.html#hybrid-esp32)

This tutorial describes how MicroPython provides an on-device filesystem, allowing standard Python file I/O methods to be used with persistent storage. MicroPython automatically creates a default configuration and auto-detects the primary filesystem, so this tutorial will be mostly useful if you want to modify the partitioning, filesystem type, or use custom block devices. The filesystem is typically backed by internal flash memory on the device, but can also use external flash, RAM, or a custom block device. On some ports (e.g. STM32), the filesystem may also be available over USB MSC to a host PC. [The pyboard.py tool](https://docs.micropython.org/en/latest/reference/pyboard.py.html#pyboard-py) also provides a way for the host PC to access to the filesystem on all ports.

Note: This is mainly for use on bare-metal ports like STM32 and ESP32. On ports with an operating system (e.g. the Unix port) the filesystem is provided by the host OS.

## VFS

MicroPython implements a Unix-like Virtual File System (VFS) layer. All mounted filesystems are combined into a single virtual filesystem, starting at the root /. Filesystems are mounted into directories in this structure, and at startup the working directory is changed to where the primary filesystem is mounted.

On STM32 / Pyboard, the internal flash is mounted at /flash, and optionally the SDCard at /sd. On ESP8266/ESP32, the primary filesystem is mounted at /.

## Block devices

A block device is an instance of a class that implements the [**uos.AbstractBlockDev**](https://docs.micropython.org/en/latest/library/uos.html#uos.AbstractBlockDev) protocol.

### [Built-in block devices](https://docs.micropython.org/en/latest/reference/filesystem.html#id5)

Ports provide built-in block devices to access their primary flash.

On power-on, MicroPython will attempt to detect the filesystem on the default flash and configure and mount it automatically. If no filesystem is found, MicroPython will attempt to create a FAT filesystem spanning the entire flash. Ports can also provide a mechanism to “factory reset” the primary flash, usually by some combination of button presses at power on.

[**STM32 / Pyboard**](https://docs.micropython.org/en/latest/reference/filesystem.html#id6)

The [pyb.Flash](https://docs.micropython.org/en/latest/library/pyb.Flash.html#pyb-flash) class provides access to the internal flash. On some boards which have larger external flash (e.g. Pyboard D), it will use that instead. The start kwarg should always be specified, i.e. pyb.Flash(start=0).

Note: For backwards compatibility, when constructed with no arguments (i.e. pyb.Flash()), it only implements the simple block interface and reflects the virtual device presented to USB MSC (i.e. it includes a virtual partition table at the start).

[**ESP8266**](https://docs.micropython.org/en/latest/reference/filesystem.html#id7)

The internal flash is exposed as a block device object which is created in the flashbdev module on start up. This object is by default added as a global variable so it can usually be accessed simply as bdev. This implements the extended interface.

[**ESP32**](https://docs.micropython.org/en/latest/reference/filesystem.html#id8)

The [esp32.Partition](https://docs.micropython.org/en/latest/library/esp32.html#esp32.Partition) class implements a block device for partitions defined for the board. Like ESP8266, there is a global variable bdev which points to the default partition. This implements the extended interface.

### [Custom block devices](https://docs.micropython.org/en/latest/reference/filesystem.html#id9)

The following class implements a simple block device that stores its data in RAM using a bytearray:

**class** **RAMBlockDev**:

**def** \_\_init\_\_(self, block\_size, num\_blocks):

self.block\_size = block\_size

self.data = bytearray(block\_size \* num\_blocks)

**def** readblocks(self, block\_num, buf):

**for** i **in** range(len(buf)):

buf[i] = self.data[block\_num \* self.block\_size + i]

**def** writeblocks(self, block\_num, buf):

**for** i **in** range(len(buf)):

self.data[block\_num \* self.block\_size + i] = buf[i]

**def** ioctl(self, op, arg):

**if** op == 4: *# get number of blocks*

**return** len(self.data) // self.block\_size

**if** op == 5: *# get block size*

**return** self.block\_size

It can be used as follows:

**import** **os**

bdev = RAMBlockDev(512, 50)

os.VfsFat.mkfs(bdev)

os.mount(bdev, '/ramdisk')

An example of a block device that supports both the simple and extended interface (i.e. both signatures and behaviours of the [uos.AbstractBlockDev.readblocks()](https://docs.micropython.org/en/latest/library/uos.html#uos.AbstractBlockDev.readblocks) and [uos.AbstractBlockDev.writeblocks()](https://docs.micropython.org/en/latest/library/uos.html#uos.AbstractBlockDev.writeblocks) methods) is:

**class** **RAMBlockDev**:

**def** \_\_init\_\_(self, block\_size, num\_blocks):

self.block\_size = block\_size

self.data = bytearray(block\_size \* num\_blocks)

**def** readblocks(self, block\_num, buf, offset=0):

addr = block\_num \* self.block\_size + offset

**for** i **in** range(len(buf)):

buf[i] = self.data[addr + i]

**def** writeblocks(self, block\_num, buf, offset=**None**):

**if** offset **is** **None**:

*# do erase, then write*

**for** i **in** range(len(buf) // self.block\_size):

self.ioctl(6, block\_num + i)

offset = 0

addr = block\_num \* self.block\_size + offset

**for** i **in** range(len(buf)):

self.data[addr + i] = buf[i]

**def** ioctl(self, op, arg):

**if** op == 4: *# block count*

**return** len(self.data) // self.block\_size

**if** op == 5: *# block size*

**return** self.block\_size

**if** op == 6: *# block erase*

**return** 0

As it supports the extended interface, it can be used with [littlefs](https://docs.micropython.org/en/latest/library/uos.html#uos.VfsLfs2):

**import** **os**

bdev = RAMBlockDev(512, 50)

os.VfsLfs2.mkfs(bdev)

os.mount(bdev, '/ramdisk')

Once mounted, the filesystem (regardless of its type) can be used as it normally would be used from Python code, for example:

**with** open('/ramdisk/hello.txt', 'w') **as** f:

f.write('Hello world')

print(open('/ramdisk/hello.txt').read())

## Filesystems

MicroPython ports can provide implementations of [FAT](https://docs.micropython.org/en/latest/library/uos.html#uos.VfsFat), [littlefs v1](https://docs.micropython.org/en/latest/library/uos.html#uos.VfsLfs1) and [littlefs v2](https://docs.micropython.org/en/latest/library/uos.html#uos.VfsLfs2).

The following table shows which filesystems are included in the firmware by default for given port/board combinations, however they can be optionally enabled in a custom firmware build.

| **Board** | **FAT** | **littlefs v1** | **littlefs v2** |
| --- | --- | --- | --- |
| pyboard 1.0, 1.1, D | Yes | No | Yes |
| Other STM32 | Yes | No | No |
| ESP8266 | Yes | No | No |
| ESP32 | Yes | No | Yes |

### [FAT](https://docs.micropython.org/en/latest/reference/filesystem.html#id11)

The main advantage of the FAT filesystem is that it can be accessed over USB MSC on supported boards (e.g. STM32) without any additional drivers required on the host PC.

However, FAT is not tolerant to power failure during writes and this can lead to filesystem corruption. For applications that do not require USB MSC, it is recommended to use littlefs instead.

To format the entire flash using FAT:

*# ESP8266 and ESP32*

**import** **os**

os.umount('/')

os.VfsFat.mkfs(bdev)

os.mount(bdev, '/')

*# STM32*

**import** **os**, **pyb**

os.umount('/flash')

os.VfsFat.mkfs(pyb.Flash(start=0))

os.mount(pyb.Flash(start=0), '/flash')

os.chdir('/flash')

### [Littlefs](https://docs.micropython.org/en/latest/reference/filesystem.html#id12)

[Littlefs](https://github.com/ARMmbed/littlefs) is a filesystem designed for flash-based devices, and is much more resistant to filesystem corruption.

There are reports of littlefs v1 and v2 failing in certain situations, for details see [littlefs issue 347](https://github.com/ARMmbed/littlefs/issues/347) and [littlefs issue 295](https://github.com/ARMmbed/littlefs/issues/295).

Note: It can be still be accessed over USB MSC using the [littlefs FUSE driver](https://github.com/ARMmbed/littlefs-fuse/tree/master/littlefs). Note that you must use the -b=4096 option to override the block size.

To format the entire flash using littlefs v2:

*# ESP8266 and ESP32*

**import** **os**

os.umount('/')

os.VfsLfs2.mkfs(bdev)

os.mount(bdev, '/')

*# STM32*

**import** **os**, **pyb**

os.umount('/flash')

os.VfsLfs2.mkfs(pyb.Flash(start=0))

os.mount(pyb.Flash(start=0), '/flash')

os.chdir('/flash')

### [Hybrid (STM32)](https://docs.micropython.org/en/latest/reference/filesystem.html#id13)

By using the start and len kwargs to **pyb.Flash**, you can create block devices spanning a subset of the flash device.

For example, to configure the first 256kiB as FAT (and available over USB MSC), and the remainder as littlefs:

**import** **os**, **pyb**

os.umount('/flash')

p1 = pyb.Flash(start=0, len=256\*1024)

p2 = pyb.Flash(start=256\*1024)

os.VfsFat.mkfs(p1)

os.VfsLfs2.mkfs(p2)

os.mount(p1, '/flash')

os.mount(p2, '/data')

os.chdir('/flash')

This might be useful to make your Python files, configuration and other rarely-modified content available over USB MSC, but allowing for frequently changing application data to reside on littlefs with better resilience to power failure, etc.

The partition at offset 0 will be mounted automatically (and the filesystem type automatically detected), but you can add:

**import** **os**, **pyb**

p2 = pyb.Flash(start=256\*1024)

os.mount(p2, '/data')

to boot.py to mount the data partition.

### [Hybrid (ESP32)](https://docs.micropython.org/en/latest/reference/filesystem.html#id14)

On ESP32, if you build custom firmware, you can modify partitions.csv to define an arbitrary partition layout.

At boot, the partition named “vfs” will be mounted at / by default, but any additional partitions can be mounted in your boot.py using:

**import** **esp32**, **os**

p = esp32.Partition.find(esp32.Partition.TYPE\_DATA, label='foo')

os.mount(p, '/foo')

# The pyboard.py tool

This is a standalone Python tool that runs on your PC that provides a way to:

* Quickly run a Python script or command on a MicroPython device. This is useful while developing MicroPython programs to quickly test code without needing to copy files to/from the device.
* Access the filesystem on a device. This allows you to deploy your code to the device (even if the board doesn’t support USB MSC).

Despite the name, pyboard.py works on all MicroPython ports that support the raw REPL (including STM32, ESP32, ESP8266, NRF).

You can download the latest version from [GitHub](https://github.com/micropython/micropython/blob/master/tools/pyboard.py). The only dependency is the pyserial library which can be installed from PiPy or your system package manager.

Running pyboard.py --help gives the following output:

usage: pyboard [-h] [--device DEVICE] [-b BAUDRATE] [-u USER]

[-p PASSWORD] [-c COMMAND] [-w WAIT] [--follow] [-f]

[files [files ...]]

Run scripts on the pyboard.

positional arguments:

files input files

optional arguments:

-h, --help show this help message and exit

--device DEVICE the serial device or the IP address of the

pyboard

-b BAUDRATE, --baudrate BAUDRATE

the baud rate of the serial device

-u USER, --user USER the telnet login username

-p PASSWORD, --password PASSWORD

the telnet login password

-c COMMAND, --command COMMAND

program passed in as string

-w WAIT, --wait WAIT seconds to wait for USB connected board to become

available

--follow follow the output after running the scripts

[default if no scripts given]

-f, --filesystem perform a filesystem action

**Running a command on the device**

This is useful for testing short snippets of code, or to script an interaction with the device.:

$ pyboard.py --device /dev/ttyACM0 -c 'print(1+1)'

2

**Running a script on the device**

If you have a script, app.py that you want to run on a device, then use:

$ pyboard.py --device /dev/ttyACM0 app.py

Note that this doesn’t actually copy app.py to the device’s filesystem, it just loads the code into RAM and executes it. Any output generated by the program will be displayed.

If the program app.py does not finish then you’ll need to stop pyboard.py, eg with Ctrl-C. The program app.py will still continue to run on the MicroPython device.

**Filesystem access**

Using the -f flag, the following filesystem operations are supported:

* cp src [src...] dest Copy files to/from the device.
* cat path Print the contents of a file on the device.
* ls [path] List contents of a directory (defaults to current working directory).
* rm path Remove a file.
* mkdir path Create a directory.
* rmdir path Remove a directory.

The cp command uses a ssh-like convention for referring to local and remote files. Any path starting with a : will be interpreted as on the device, otherwise it will be local. So:

$ pyboard.py --device /dev/ttyACM0 -f cp main.py :main.py

will copy main.py from the current directory on the PC to a file named main.py on the device. The filename can be omitted, e.g.:

$ pyboard.py --device /dev/ttyACM0 -f cp main.py :

is equivalent to the above.

Some more examples:

# Copy main.py from the device to the local PC.

$ pyboard.py --device /dev/ttyACM0 -f cp :main.py main.py

# Same, but using . instead.

$ pyboard.py --device /dev/ttyACM0 -f cp :main.py .

# Copy three files to the device, keeping their names

# and paths (note: `lib` must exist on the device)

$ pyboard.py --device /dev/ttyACM0 -f cp main.py app.py lib/foo.py :

# Remove a file from the device.

$ pyboard.py --device /dev/ttyACM0 -f rm util.py

# Print the contents of a file on the device.

$ pyboard.py --device /dev/ttyACM0 -f cat boot.py

...contents of boot.py...

**Using the pyboard library**

You can also use pyboard.py as a library for scripting interactions with a MicroPython board.

**import** **pyboard**

pyb = pyboard.Pyboard('/dev/ttyACM0', 115200)

pyb.enter\_raw\_repl()

ret = pyb.**exec**('print(1+1)')

**print**(ret)

pyb.exit\_raw\_repl()

# MicroPython libraries

Warning

Important summary of this section

* MicroPython implements a subset of Python functionality for each module.
* To ease extensibility, MicroPython versions of standard Python modules usually have u (“micro”) prefix.
* Any particular MicroPython variant or port may miss any feature/function described in this general documentation (due to resource constraints or other limitations).

This chapter describes modules (function and class libraries) which are built into MicroPython.

There are a few categories of such modules:

* Modules which implement a subset of standard Python functionality and are not intended to be extended by the user.
* Modules which implement a subset of Python functionality, with a provision for extension by the user (via Python code).
* Modules which implement MicroPython extensions to the Python standard libraries.
* Modules specific to a particular [**MicroPython port**](https://docs.micropython.org/en/latest/reference/glossary.html#term-micropython-port) and thus not portable.

Note about the availability of the modules and their contents: This documentation in general aspires to describe all modules and functions/classes which are implemented in MicroPython project. However, MicroPython is highly configurable, and each port to a particular board/embedded system makes available only a subset of MicroPython libraries. For officially supported ports, there is an effort to either filter out non-applicable items, or mark individual descriptions with “Availability:” clauses describing which ports provide a given feature.

With that in mind, please still be warned that some functions/classes in a module (or even the entire module) described in this documentation **may be unavailable** in a particular build of MicroPython on a particular system. The best place to find general information of the availability/non-availability of a particular feature is the “General Information” section which contains information pertaining to a specific [**MicroPython port**](https://docs.micropython.org/en/latest/reference/glossary.html#term-micropython-port).

On some ports you are able to discover the available, built-in libraries that can be imported by entering the following at the REPL:

help('modules')

Beyond the built-in libraries described in this documentation, many more modules from the Python standard library, as well as further MicroPython extensions to it, can be found in [**micropython-lib**](https://docs.micropython.org/en/latest/reference/glossary.html#term-micropython-lib).

**Python standard libraries and micro-libraries**

The following standard Python libraries have been “micro-ified” to fit in with the philosophy of MicroPython. They provide the core functionality of that module and are intended to be a drop-in replacement for the standard Python library. Some modules below use a standard Python name, but prefixed with “u”, e.g. ujson instead of json. This is to signify that such a module is micro-library, i.e. implements only a subset of CPython module functionality. By naming them differently, a user has a choice to write a Python-level module to extend functionality for better compatibility with CPython (indeed, this is what done by the [micropython-lib](https://docs.micropython.org/en/latest/reference/glossary.html#term-micropython-lib) project mentioned above).

On some embedded platforms, where it may be cumbersome to add Python-level wrapper modules to achieve naming compatibility with CPython, micro-modules are available both by their u-name, and also by their non-u-name. The non-u-name can be overridden by a file of that name in your library path (sys.path). For example, import json will first search for a file json.py (or package directory json) and load that module if it is found. If nothing is found, it will fallback to loading the built-in ujson module.

* [Builtin functions and exceptions](https://docs.micropython.org/en/latest/library/builtins.html)
* [cmath – mathematical functions for complex numbers](https://docs.micropython.org/en/latest/library/cmath.html)
* [gc – control the garbage collector](https://docs.micropython.org/en/latest/library/gc.html)
* [math – mathematical functions](https://docs.micropython.org/en/latest/library/math.html)
* [sys – system specific functions](https://docs.micropython.org/en/latest/library/sys.html)
* [uarray – arrays of numeric data](https://docs.micropython.org/en/latest/library/uarray.html)
* [ubinascii – binary/ASCII conversions](https://docs.micropython.org/en/latest/library/ubinascii.html)
* [ucollections – collection and container types](https://docs.micropython.org/en/latest/library/ucollections.html)
* [uerrno – system error codes](https://docs.micropython.org/en/latest/library/uerrno.html)
* [uhashlib – hashing algorithms](https://docs.micropython.org/en/latest/library/uhashlib.html)
* [uheapq – heap queue algorithm](https://docs.micropython.org/en/latest/library/uheapq.html)
* [uio – input/output streams](https://docs.micropython.org/en/latest/library/uio.html)
* [ujson – JSON encoding and decoding](https://docs.micropython.org/en/latest/library/ujson.html)
* [uos – basic “operating system” services](https://docs.micropython.org/en/latest/library/uos.html)
* [ure – simple regular expressions](https://docs.micropython.org/en/latest/library/ure.html)
* [uselect – wait for events on a set of streams](https://docs.micropython.org/en/latest/library/uselect.html)
* [usocket – socket module](https://docs.micropython.org/en/latest/library/usocket.html)
* [ussl – SSL/TLS module](https://docs.micropython.org/en/latest/library/ussl.html)
* [ustruct – pack and unpack primitive data types](https://docs.micropython.org/en/latest/library/ustruct.html)
* [utime – time related functions](https://docs.micropython.org/en/latest/library/utime.html)
* [uzlib – zlib decompression](https://docs.micropython.org/en/latest/library/uzlib.html)
* [\_thread – multithreading support](https://docs.micropython.org/en/latest/library/_thread.html)

**MicroPython-specific libraries**

Functionality specific to the MicroPython implementation is available in the following libraries.

* [btree – simple BTree database](https://docs.micropython.org/en/latest/library/btree.html)
* [framebuf — frame buffer manipulation](https://docs.micropython.org/en/latest/library/framebuf.html)
* [machine — functions related to the hardware](https://docs.micropython.org/en/latest/library/machine.html)
* [micropython – access and control MicroPython internals](https://docs.micropython.org/en/latest/library/micropython.html)
* [network — network configuration](https://docs.micropython.org/en/latest/library/network.html)
* [ubluetooth — low-level Bluetooth](https://docs.micropython.org/en/latest/library/ubluetooth.html)
* [ucryptolib – cryptographic ciphers](https://docs.micropython.org/en/latest/library/ucryptolib.html)
* [uctypes – access binary data in a structured way](https://docs.micropython.org/en/latest/library/uctypes.html)

**Port-specific libraries**

In some cases the following port/board-specific libraries have functions or classes similar to those in the [machine](https://docs.micropython.org/en/latest/library/machine.html#module-machine) library. Where this occurs, the entry in the port specific library exposes hardware functionality unique to that platform.

To write portable code use functions and classes from the [machine](https://docs.micropython.org/en/latest/library/machine.html#module-machine) module. To access platform-specific hardware use the appropriate library, e.g. [pyb](https://docs.micropython.org/en/latest/library/pyb.html#module-pyb) in the case of the Pyboard.

**Libraries specific to the pyboard**

The following libraries are specific to the pyboard.

* [pyb — functions related to the board](https://docs.micropython.org/en/latest/library/pyb.html)
* [Time related functions](https://docs.micropython.org/en/latest/library/pyb.html#time-related-functions)
* [Reset related functions](https://docs.micropython.org/en/latest/library/pyb.html#reset-related-functions)
* [Interrupt related functions](https://docs.micropython.org/en/latest/library/pyb.html#interrupt-related-functions)
* [Power related functions](https://docs.micropython.org/en/latest/library/pyb.html#power-related-functions)
* [Miscellaneous functions](https://docs.micropython.org/en/latest/library/pyb.html#miscellaneous-functions)
* [Classes](https://docs.micropython.org/en/latest/library/pyb.html#classes)
* [lcd160cr — control of LCD160CR display](https://docs.micropython.org/en/latest/library/lcd160cr.html)
* [class LCD160CR](https://docs.micropython.org/en/latest/library/lcd160cr.html#class-lcd160cr)
* [Constructors](https://docs.micropython.org/en/latest/library/lcd160cr.html#constructors)
* [Static methods](https://docs.micropython.org/en/latest/library/lcd160cr.html#static-methods)
* [Instance members](https://docs.micropython.org/en/latest/library/lcd160cr.html#instance-members)
* [Setup commands](https://docs.micropython.org/en/latest/library/lcd160cr.html#setup-commands)
* [Pixel access methods](https://docs.micropython.org/en/latest/library/lcd160cr.html#pixel-access-methods)
* [Drawing text](https://docs.micropython.org/en/latest/library/lcd160cr.html#drawing-text)
* [Drawing primitive shapes](https://docs.micropython.org/en/latest/library/lcd160cr.html#drawing-primitive-shapes)
* [Touch screen methods](https://docs.micropython.org/en/latest/library/lcd160cr.html#touch-screen-methods)
* [Advanced commands](https://docs.micropython.org/en/latest/library/lcd160cr.html#advanced-commands)
* [Constants](https://docs.micropython.org/en/latest/library/lcd160cr.html#constants)

**Libraries specific to the WiPy**

The following libraries and classes are specific to the WiPy.

* [wipy – WiPy specific features](https://docs.micropython.org/en/latest/library/wipy.html)
* [Functions](https://docs.micropython.org/en/latest/library/wipy.html#functions)
* [class ADCWiPy – analog to digital conversion](https://docs.micropython.org/en/latest/library/machine.ADCWiPy.html)
* [Constructors](https://docs.micropython.org/en/latest/library/machine.ADCWiPy.html#constructors)
* [Methods](https://docs.micropython.org/en/latest/library/machine.ADCWiPy.html#methods)
* [class ADCChannel — read analog values from internal or external sources](https://docs.micropython.org/en/latest/library/machine.ADCWiPy.html#class-adcchannel-read-analog-values-from-internal-or-external-sources)
* [class TimerWiPy – control hardware timers](https://docs.micropython.org/en/latest/library/machine.TimerWiPy.html)
* [Constructors](https://docs.micropython.org/en/latest/library/machine.TimerWiPy.html#constructors)
* [Methods](https://docs.micropython.org/en/latest/library/machine.TimerWiPy.html#methods)
* [class TimerChannel — setup a channel for a timer](https://docs.micropython.org/en/latest/library/machine.TimerWiPy.html#class-timerchannel-setup-a-channel-for-a-timer)
* [Methods](https://docs.micropython.org/en/latest/library/machine.TimerWiPy.html#id1)
* [Constants](https://docs.micropython.org/en/latest/library/machine.TimerWiPy.html#constants)

**Libraries specific to the ESP8266 and ESP32**

The following libraries are specific to the ESP8266 and ESP32.

* [esp — functions related to the ESP8266 and ESP32](https://docs.micropython.org/en/latest/library/esp.html)
* [Functions](https://docs.micropython.org/en/latest/library/esp.html#functions)
* [esp32 — functionality specific to the ESP32](https://docs.micropython.org/en/latest/library/esp32.html)
* [Functions](https://docs.micropython.org/en/latest/library/esp32.html#functions)
* [Flash partitions](https://docs.micropython.org/en/latest/library/esp32.html#flash-partitions)
* [RMT](https://docs.micropython.org/en/latest/library/esp32.html#rmt)
* [Ultra-Low-Power co-processor](https://docs.micropython.org/en/latest/library/esp32.html#ultra-low-power-co-processor)
* [Constants](https://docs.micropython.org/en/latest/library/esp32.html#id1)

# MicroPython differences from CPython

The operations listed in this section produce conflicting results in MicroPython when compared to standard Python. MicroPython implements Python 3.4 and some select features of Python 3.5.

* [Syntax](https://docs.micropython.org/en/latest/genrst/syntax.html)
* [Spaces](https://docs.micropython.org/en/latest/genrst/syntax.html#spaces)
* [Unicode](https://docs.micropython.org/en/latest/genrst/syntax.html#unicode)
* [Core language](https://docs.micropython.org/en/latest/genrst/core_language.html)
* [Classes](https://docs.micropython.org/en/latest/genrst/core_language.html#classes)
* [Functions](https://docs.micropython.org/en/latest/genrst/core_language.html#functions)
* [Generator](https://docs.micropython.org/en/latest/genrst/core_language.html#generator)
* [Runtime](https://docs.micropython.org/en/latest/genrst/core_language.html#runtime)
* [import](https://docs.micropython.org/en/latest/genrst/core_language.html#import)
* [Builtin types](https://docs.micropython.org/en/latest/genrst/builtin_types.html)
* [Exception](https://docs.micropython.org/en/latest/genrst/builtin_types.html#exception)
* [bytearray](https://docs.micropython.org/en/latest/genrst/builtin_types.html#bytearray)
* [bytes](https://docs.micropython.org/en/latest/genrst/builtin_types.html#bytes)
* [dict](https://docs.micropython.org/en/latest/genrst/builtin_types.html#dict)
* [float](https://docs.micropython.org/en/latest/genrst/builtin_types.html#float)
* [int](https://docs.micropython.org/en/latest/genrst/builtin_types.html#int)
* [list](https://docs.micropython.org/en/latest/genrst/builtin_types.html#list)
* [str](https://docs.micropython.org/en/latest/genrst/builtin_types.html#str)
* [tuple](https://docs.micropython.org/en/latest/genrst/builtin_types.html#tuple)
* [Modules](https://docs.micropython.org/en/latest/genrst/modules.html)
* [array](https://docs.micropython.org/en/latest/genrst/modules.html#array)
* [builtins](https://docs.micropython.org/en/latest/genrst/modules.html#builtins)
* [deque](https://docs.micropython.org/en/latest/genrst/modules.html#deque)
* [json](https://docs.micropython.org/en/latest/genrst/modules.html#json)
* [struct](https://docs.micropython.org/en/latest/genrst/modules.html#struct)
* [sys](https://docs.micropython.org/en/latest/genrst/modules.html#sys)

# Syntax

Generated Thu 09 Jan 2020 12:08:52 UTC

## Spaces

### uPy requires spaces between literal numbers and keywords, CPy doesn’t

Sample code:

**try**:

print(eval('1and 0'))

**except** SyntaxError:

print('Should have worked')

**try**:

print(eval('1or 0'))

**except** SyntaxError:

print('Should have worked')

**try**:

print(eval('1if 1else 0'))

**except** SyntaxError:

print('Should have worked')

|  |  |
| --- | --- |
| CPy output: | uPy output: |
| 0  1  1 | Should have worked  Should have worked  Should have worked |

## Unicode

### Unicode name escapes are not implemented

Sample code:

print("**\N{LATIN SMALL LETTER A}**")

|  |  |
| --- | --- |
| CPy output: | uPy output: |
| a | NotImplementedError: unicode name escapes |

# Core language

Generated Thu 09 Jan 2020 12:08:52 UTC

## Classes

### Special method \_\_del\_\_ not implemented for user-defined classes

Sample code:

**import** **gc**

**class** **Foo**():

**def** \_\_del\_\_(self):

print('\_\_del\_\_')

f = Foo()

**del** f

gc.collect()

|  |  |
| --- | --- |
| CPy output: | uPy output: |
| \_\_del\_\_ |  |

### Method Resolution Order (MRO) is not compliant with CPython

**Cause:** Depth first non-exhaustive method resolution order

**Workaround:** Avoid complex class hierarchies with multiple inheritance and complex method overrides. Keep in mind that many languages don’t support multiple inheritance at all.

Sample code:

**class** **Foo**:

**def** \_\_str\_\_(self):

**return** "Foo"

**class** **C**(tuple, Foo):

**pass**

t = C((1, 2, 3))

print(t)

|  |  |
| --- | --- |
| CPy output: | uPy output: |
| Foo | (1, 2, 3) |

### When inheriting from multiple classes super() only calls one class

**Cause:** See [Method Resolution Order (MRO) is not compliant with CPython](https://docs.micropython.org/en/latest/genrst/core_language.html#cpydiff-core-class-mro)

**Workaround:** See [Method Resolution Order (MRO) is not compliant with CPython](https://docs.micropython.org/en/latest/genrst/core_language.html#cpydiff-core-class-mro)

Sample code:

**class** **A**:

**def** \_\_init\_\_(self):

print("A.\_\_init\_\_")

**class** **B**(A):

**def** \_\_init\_\_(self):

print("B.\_\_init\_\_")

super().\_\_init\_\_()

**class** **C**(A):

**def** \_\_init\_\_(self):

print("C.\_\_init\_\_")

super().\_\_init\_\_()

**class** **D**(B,C):

**def** \_\_init\_\_(self):

print("D.\_\_init\_\_")

super().\_\_init\_\_()

D()

|  |  |
| --- | --- |
| CPy output: | uPy output: |
| D.\_\_init\_\_  B.\_\_init\_\_  C.\_\_init\_\_  A.\_\_init\_\_ | D.\_\_init\_\_  B.\_\_init\_\_  A.\_\_init\_\_ |

### Calling super() getter property in subclass will return a property object, not the value

Sample code:

**class** **A**:

**@property**

**def** p(self):

**return** {"a":10}

**class** **AA**(A):

**@property**

**def** p(self):

**return** super().p

a = AA()

print(a.p)

|  |  |
| --- | --- |
| CPy output: | uPy output: |
| {'a': 10} | <property> |

## Functions

### Error messages for methods may display unexpected argument counts

**Cause:** MicroPython counts “self” as an argument.

**Workaround:** Interpret error messages with the information above in mind.

Sample code:

**try**:

[].append()

**except** Exception **as** e:

print(e)

|  |  |
| --- | --- |
| CPy output: | uPy output: |
| append() takes exactly one argument (0 given) | function takes 2 positional arguments but 1 were given |

### User-defined attributes for functions are not supported

**Cause:** MicroPython is highly optimized for memory usage.

**Workaround:** Use external dictionary, e.g. FUNC\_X[f] = 0.

Sample code:

**def** f():

**pass**

f.x = 0

print(f.x)

|  |  |
| --- | --- |
| CPy output: | uPy output: |
| 0 | Traceback (most recent call last):  File "<stdin>", line 10, **in** <module>  AttributeError: 'function' object has no attribute 'x' |

## Generator

### Context manager \_\_exit\_\_() not called in a generator which does not run to completion

Sample code:

**class** **foo**(object):

**def** \_\_enter\_\_(self):

print('Enter')

**def** \_\_exit\_\_(self, \*args):

print('Exit')

**def** bar(x):

**with** foo():

**while** **True**:

x += 1

**yield** x

**def** func():

g = bar(0)

**for** \_ **in** range(3):

print(next(g))

func()

|  |  |
| --- | --- |
| CPy output: | uPy output: |
| Enter  1  2  3  Exit | Enter  1  2  3 |

## Runtime

### Local variables aren’t included in locals() result

**Cause:** MicroPython doesn’t maintain symbolic local environment, it is optimized to an array of slots. Thus, local variables can’t be accessed by a name.

Sample code:

**def** test():

val = 2

print(locals())

test()

|  |  |
| --- | --- |
| CPy output: | uPy output: |
| {'val': 2} | {'test': <function test at 0x7f3490fa0100>, '\_\_name\_\_': '\_\_main\_\_', '\_\_file\_\_': '<stdin>'} |

### Code running in eval() function doesn’t have access to local variables

**Cause:** MicroPython doesn’t maintain symbolic local environment, it is optimized to an array of slots. Thus, local variables can’t be accessed by a name. Effectively, eval(expr) in MicroPython is equivalent to eval(expr, globals(), globals()).

Sample code:

val = 1

**def** test():

val = 2

print(val)

eval("print(val)")

test()

|  |  |
| --- | --- |
| CPy output: | uPy output: |
| 2  2 | 2  1 |

## import

### \_\_path\_\_ attribute of a package has a different type (single string instead of list of strings) in MicroPython

**Cause:** MicroPython does’t support namespace packages split across filesystem. Beyond that, MicroPython’s import system is highly optimized for minimal memory usage.

**Workaround:** Details of import handling is inherently implementation dependent. Don’t rely on such details in portable applications.

Sample code:

**import** **modules**

print(modules.\_\_path\_\_)

|  |  |
| --- | --- |
| CPy output: | uPy output: |
| ['/home/micropython/micropython-docs/tests/cpydiff/modules'] | ../tests/cpydiff//modules |

### Failed to load modules are still registered as loaded

**Cause:** To make module handling more efficient, it’s not wrapped with exception handling.

**Workaround:** Test modules before production use; during development, use del sys.modules["name"], or just soft or hard reset the board.

Sample code:

**import** **sys**

**try**:

**from** **modules** **import** foo

**except** NameError **as** e:

print(e)

**try**:

**from** **modules** **import** foo

print('Should not get here')

**except** NameError **as** e:

print(e)

|  |  |
| --- | --- |
| CPy output: | uPy output: |
| foo  name 'xxx' **is** **not** defined  foo  name 'xxx' **is** **not** defined | foo  name 'xxx' isn't defined  Should **not** get here |

### MicroPython does’t support namespace packages split across filesystem.

**Cause:** MicroPython’s import system is highly optimized for simplicity, minimal memory usage, and minimal filesystem search overhead.

**Workaround:** Don’t install modules belonging to the same namespace package in different directories. For MicroPython, it’s recommended to have at most 3-component module search paths: for your current application, per-user (writable), system-wide (non-writable).

Sample code:

**import** **sys**

sys.path.append(sys.path[1] + "/modules")

sys.path.append(sys.path[1] + "/modules2")

**import** **subpkg.foo**

**import** **subpkg.bar**

print("Two modules of a split namespace package imported")

|  |  |
| --- | --- |
| CPy output: | uPy output: |
| Two modules of a split namespace package imported | Traceback (most recent call last):  File "<stdin>", line 12, **in** <module>  ImportError: no module named 'subpkg.bar' |

# Builtin types

Generated Thu 09 Jan 2020 12:08:52 UTC

## Exception

### Exception chaining not implemented

Sample code:

**try**:

**raise** TypeError

**except** TypeError:

**raise** ValueError

|  |  |
| --- | --- |
| CPy output: | uPy output: |
| Traceback (most recent call last):  File "<stdin>", line 8, **in** <module>  TypeError  During handling of the above exception, another exception occurred:  Traceback (most recent call last):  File "<stdin>", line 10, **in** <module>  ValueError | Traceback (most recent call last):  File "<stdin>", line 10, **in** <module>  ValueError: |

### User-defined attributes for builtin exceptions are not supported

**Cause:** MicroPython is highly optimized for memory usage.

**Workaround:** Use user-defined exception subclasses.

Sample code:

e = Exception()

e.x = 0

print(e.x)

|  |  |
| --- | --- |
| CPy output: | uPy output: |
| 0 | Traceback (most recent call last):  File "<stdin>", line 8, **in** <module>  AttributeError: 'Exception' object has no attribute 'x' |

### Exception in while loop condition may have unexpected line number

**Cause:** Condition checks are optimized to happen at the end of loop body, and that line number is reported.

Sample code:

l = ["-foo", "-bar"]

i = 0

**while** l[i][0] == "-":

print("iter")

i += 1

|  |  |
| --- | --- |
| CPy output: | uPy output: |
| iter  iter  Traceback (most recent call last):  File "<stdin>", line 10, **in** <module>  IndexError: list index out of range | iter  iter  Traceback (most recent call last):  File "<stdin>", line 12, **in** <module>  IndexError: list index out of range |

### Exception.\_\_init\_\_ method does not exist.

**Cause:** Subclassing native classes is not fully supported in MicroPython.

**Workaround:** Call using super() instead:

**class** **A**(Exception):

**def** \_\_init\_\_(self):

super().\_\_init\_\_()

Sample code:

**class** **A**(Exception):

**def** \_\_init\_\_(self):

Exception.\_\_init\_\_(self)

a = A()

|  |  |
| --- | --- |
| CPy output: | uPy output: |
|  | Traceback (most recent call last):  File "<stdin>", line 15, **in** <module>  File "<stdin>", line 13, **in** \_\_init\_\_  AttributeError: type object 'Exception' has no attribute '\_\_init\_\_' |

## bytearray

### Array slice assignment with unsupported RHS

Sample code:

b = bytearray(4)

b[0:1] = [1, 2]

print(b)

|  |  |
| --- | --- |
| CPy output: | uPy output: |
| bytearray(b'**\x01\x02\x00\x00\x00**') | Traceback (most recent call last):  File "<stdin>", line 8, **in** <module>  NotImplementedError: array/bytes required on right side |

## bytes

### bytes objects support .format() method

**Cause:** MicroPython strives to be a more regular implementation, so if both [**str**](https://docs.micropython.org/en/latest/library/builtins.html#str) and [**bytes**](https://docs.micropython.org/en/latest/library/builtins.html#bytes) support \_\_mod\_\_() (the % operator), it makes sense to support format() for both too. Support for \_\_mod\_\_ can also be compiled out, which leaves only format() for bytes formatting.

**Workaround:** If you are interested in CPython compatibility, don’t use .format() on bytes objects.

Sample code:

print(b'*{}*'.format(1))

|  |  |
| --- | --- |
| CPy output: | uPy output: |
| Traceback (most recent call last):  File "<stdin>", line 7, **in** <module>  AttributeError: 'bytes' object has no attribute 'format' | b'1' |

### bytes() with keywords not implemented

**Workaround:** Pass the encoding as a positional parameter, e.g. print(bytes('abc', 'utf-8'))

Sample code:

print(bytes('abc', encoding='utf8'))

|  |  |
| --- | --- |
| CPy output: | uPy output: |
| b'abc' | Traceback (most recent call last):  File "<stdin>", line 7, **in** <module>  NotImplementedError: keyword argument(s) **not** yet implemented - use normal args instead |

### Bytes subscription with step != 1 not implemented

**Cause:** MicroPython is highly optimized for memory usage.

**Workaround:** Use explicit loop for this very rare operation.

Sample code:

print(b'123'[0:3:2])

|  |  |
| --- | --- |
| CPy output: | uPy output: |
| b'13' | Traceback (most recent call last):  File "<stdin>", line 7, **in** <module>  NotImplementedError: only slices **with** step=1 (aka **None**) are supported |

## dict

### Dictionary keys view does not behave as a set.

**Cause:** Not implemented.

**Workaround:** Explicitly convert keys to a set before using set operations.

Sample code:

print({1:2, 3:4}.keys() & {1})

|  |  |
| --- | --- |
| CPy output: | uPy output: |
| {1} | Traceback (most recent call last):  File "<stdin>", line 7, **in** <module>  TypeError: unsupported types **for** \_\_and\_\_: 'dict\_view', 'set' |

## float

### uPy and CPython outputs formats may differ

Sample code:

print('*%.1g*' % -9.9)

|  |  |
| --- | --- |
| CPy output: | uPy output: |
| -1e+01 | -10 |

## int

### No int conversion for int-derived types available

**Workaround:** Avoid subclassing builtin types unless really needed. Prefer <https://en.wikipedia.org/wiki/Composition_over_inheritance> .

Sample code:

**class** **A**(int):

\_\_add\_\_ = **lambda** self, other: A(int(self) + other)

a = A(42)

print(a+a)

|  |  |
| --- | --- |
| CPy output: | uPy output: |
| 84 | Traceback (most recent call last):  File "<stdin>", line 11, **in** <module>  File "<stdin>", line 8, **in** <**lambda**>  TypeError: unsupported types **for** \_\_radd\_\_: 'int', 'int' |

## list

### List delete with step != 1 not implemented

**Workaround:** Use explicit loop for this rare operation.

Sample code:

l = [1, 2, 3, 4]

**del** l[0:4:2]

print(l)

|  |  |
| --- | --- |
| CPy output: | uPy output: |
| [2, 4] | Traceback (most recent call last):  File "<stdin>", line 8, **in** <module>  NotImplementedError: |

### List slice-store with non-iterable on RHS is not implemented

**Cause:** RHS is restricted to be a tuple or list

**Workaround:** Use list(<iter>) on RHS to convert the iterable to a list

Sample code:

l = [10, 20]

l[0:1] = range(4)

print(l)

|  |  |
| --- | --- |
| CPy output: | uPy output: |
| [0, 1, 2, 3, 20] | Traceback (most recent call last):  File "<stdin>", line 8, **in** <module>  TypeError: object 'range' isn't a tuple or list |

### List store with step != 1 not implemented

**Workaround:** Use explicit loop for this rare operation.

Sample code:

l = [1, 2, 3, 4]

l[0:4:2] = [5, 6]

print(l)

|  |  |
| --- | --- |
| CPy output: | uPy output: |
| [5, 2, 6, 4] | Traceback (most recent call last):  File "<stdin>", line 8, **in** <module>  NotImplementedError: |

## str

### Start/end indices such as str.endswith(s, start) not implemented

Sample code:

print('abc'.endswith('c', 1))

|  |  |
| --- | --- |
| CPy output: | uPy output: |
| **True** | Traceback (most recent call last):  File "<stdin>", line 7, **in** <module>  NotImplementedError: start/end indices |

### Attributes/subscr not implemented

Sample code:

print('*{a[0]}*'.format(a=[1, 2]))

|  |  |
| --- | --- |
| CPy output: | uPy output: |
| 1 | Traceback (most recent call last):  File "<stdin>", line 7, **in** <module>  NotImplementedError: attributes **not** supported yet |

### str(...) with keywords not implemented

**Workaround:** Input the encoding format directly. eg print(bytes('abc', 'utf-8'))

Sample code:

print(str(b'abc', encoding='utf8'))

|  |  |
| --- | --- |
| CPy output: | uPy output: |
| abc | Traceback (most recent call last):  File "<stdin>", line 7, **in** <module>  NotImplementedError: keyword argument(s) **not** yet implemented - use normal args instead |

### str.ljust() and str.rjust() not implemented

**Cause:** MicroPython is highly optimized for memory usage. Easy workarounds available.

**Workaround:** Instead of s.ljust(10) use "%-10s" % s, instead of s.rjust(10) use "% 10s" % s. Alternatively, "{:<10}".format(s) or "{:>10}".format(s).

Sample code:

print('abc'.ljust(10))

|  |  |
| --- | --- |
| CPy output: | uPy output: |
| abc | Traceback (most recent call last):  File "<stdin>", line 7, **in** <module>  AttributeError: 'str' object has no attribute 'ljust' |

### None as first argument for rsplit such as str.rsplit(None, n) not implemented

Sample code:

print('a a a'.rsplit(**None**, 1))

|  |  |
| --- | --- |
| CPy output: | uPy output: |
| ['a a', 'a'] | Traceback (most recent call last):  File "<stdin>", line 7, **in** <module>  NotImplementedError: rsplit(**None**,n) |

### Instance of a subclass of str cannot be compared for equality with an instance of a str

Sample code:

**class** **S**(str):

**pass**

s = S('hello')

print(s == 'hello')

|  |  |
| --- | --- |
| CPy output: | uPy output: |
| **True** | **False** |

### Subscript with step != 1 is not yet implemented

Sample code:

print('abcdefghi'[0:9:2])

|  |  |
| --- | --- |
| CPy output: | uPy output: |
| acegi | Traceback (most recent call last):  File "<stdin>", line 7, **in** <module>  NotImplementedError: only slices **with** step=1 (aka **None**) are supported |

## tuple

### Tuple load with step != 1 not implemented

Sample code:

print((1, 2, 3, 4)[0:4:2])

|  |  |
| --- | --- |
| CPy output: | uPy output: |
| (1, 3) | Traceback (most recent call last):  File "<stdin>", line 7, **in** <module>  NotImplementedError: only slices **with** step=1 (aka **None**) are supported |

# Modules

Generated Thu 09 Jan 2020 12:08:52 UTC

## array

### Looking for integer not implemented

Sample code:

**import** **array**

print(1 **in** array.array('B', b'12'))

|  |  |
| --- | --- |
| CPy output: | uPy output: |
| **False** | Traceback (most recent call last):  File "<stdin>", line 8, **in** <module>  NotImplementedError: |

### Array deletion not implemented

Sample code:

**import** **array**

a = array.array('b', (1, 2, 3))

**del** a[1]

print(a)

|  |  |
| --- | --- |
| CPy output: | uPy output: |
| array('b', [1, 3]) | Traceback (most recent call last):  File "<stdin>", line 9, **in** <module>  TypeError: 'array' object doesn't support item deletion |

### Subscript with step != 1 is not yet implemented

Sample code:

**import** **array**

a = array.array('b', (1, 2, 3))

print(a[3:2:2])

|  |  |
| --- | --- |
| CPy output: | uPy output: |
| array('b') | Traceback (most recent call last):  File "<stdin>", line 9, **in** <module>  NotImplementedError: only slices **with** step=1 (aka **None**) are supported |

## builtins

### Second argument to next() is not implemented

**Cause:** MicroPython is optimised for code space.

**Workaround:** Instead of val = next(it, deflt) use:

**try**:

val = next(it)

**except** StopIteration:

val = deflt

Sample code:

print(next(iter(range(0)), 42))

|  |  |
| --- | --- |
| CPy output: | uPy output: |
| 42 | Traceback (most recent call last):  File "<stdin>", line 12, **in** <module>  TypeError: function takes 1 positional arguments but 2 were given |

## deque

### Deque not implemented

**Workaround:** Use regular lists. micropython-lib has implementation of collections.deque.

Sample code:

**import** **collections**

D = collections.deque()

print(D)

|  |  |
| --- | --- |
| CPy output: | uPy output: |
| deque([]) | Traceback (most recent call last):  File "<stdin>", line 8, **in** <module>  TypeError: function missing 2 required positional arguments |

## json

### JSON module does not throw exception when object is not serialisable

Sample code:

**import** **json**

a = bytes(x **for** x **in** range(256))

**try**:

z = json.dumps(a)

x = json.loads(z)

print('Should not get here')

**except** TypeError:

print('TypeError')

|  |  |
| --- | --- |
| CPy output: | uPy output: |
| TypeError | Should **not** get here |

## struct

### Struct pack with too few args, not checked by uPy

Sample code:

**import** **struct**

**try**:

print(struct.pack('bb', 1))

print('Should not get here')

**except**:

print('struct.error')

|  |  |
| --- | --- |
| CPy output: | uPy output: |
| struct.error | b'**\x01\x00**'  Should **not** get here |

### Struct pack with too many args, not checked by uPy

Sample code:

**import** **struct**

**try**:

print(struct.pack('bb', 1, 2, 3))

print('Should not get here')

**except**:

print('struct.error')

|  |  |
| --- | --- |
| CPy output: | uPy output: |
| struct.error | b'**\x01\x02**'  Should **not** get here |

## sys

### Overriding sys.stdin, sys.stdout and sys.stderr not possible

**Cause:** They are stored in read-only memory.

Sample code:

**import** **sys**

sys.stdin = **None**

print(sys.stdin)

|  |  |
| --- | --- |
| CPy output: | uPy output: |
| **None** | Traceback (most recent call last):  File "<stdin>", line 8, **in** <module>  AttributeError: 'module' object has no attribute 'stdin' |