This is the documentation for the latest development branch of MicroPython and may refer to features that are not available in released versions.

If you are looking for the documentation for a specific release, use the drop-down menu on the left and select the desired version.

# `socket` – socket module

*This module implements a subset of the corresponding CPython module, as described below. For more information, refer to the original CPython documentation:* `socket`.

This module provides access to the BSD socket interface.

**Difference to CPython**

For efficiency and consistency, socket objects in MicroPython implement a `stream` (file-like) interface directly. In CPython, you need to convert a socket to a file-like object using `makefile()` method. This method is still supported by MicroPython (but is a no-op), so where compatibility with CPython matters, be sure to use it.

## Socket address format(s)

The native socket address format of the `socket` module is an opaque data type returned by `getaddrinfo` function, which must be used to resolve textual address (including numeric addresses):

```
sockaddr = socket.getaddrinfo('www.micropython.org', 80)[0][-1]
# You must use getaddrinfo() even for numeric addresses
sockaddr = socket.getaddrinfo('127.0.0.1', 80)[0][-1]
# Now you can use that address
sock.connect(sockaddr)
```

Using `getaddrinfo` is the most efficient (both in terms of memory and processing power) and portable way to work with addresses.

However, `socket` module (note the difference with native MicroPython `socket` module described here) provides CPython-compatible way to specify addresses using tuples, as described below. Note that depending on a MicroPython port, `socket` module can be builtin or need to be installed from `micropython-lib` (as in the case of MicroPython Unix port), and some ports still accept only numeric addresses in the tuple format, and require to use `getaddrinfo` function to resolve domain names.

Summing up:

- Always use `getaddrinfo` when writing portable applications.
- Tuple addresses described below can be used as a shortcut for quick hacks and interactive use, if your port supports them.

Tuple address format for `socket` module:

- IPv4: *(ipv4_address, port)*, where *ipv4_address* is a string with dot-notation numeric IPv4 address, e.g. `"8.8.8.8"`, and *port* is and integer port number in the range 1-65535. Note the domain names are not accepted as *ipv4_address*, they should be resolved first using `socket.getaddrinfo()`.
- IPv6: *(ipv6_address, port, flowinfo, scopeid)*, where *ipv6_address* is a string with colon-notation numeric IPv6 address, e.g. `"2001:db8::1"`, and *port* is an integer port number in the range 1-65535. *flowinfo* must be 0. *scopeid* is the interface scope identifier for link-local addresses. Note the domain names are not accepted as *ipv6_address*, they should be resolved first using `socket.getaddrinfo()`. Availability of IPv6 support depends on a MicroPython port.

## Functions

**socket.getaddrinfo**(*host, port, af=0, type=0, proto=0, flags=0, /*)

Translate the host/port argument into a sequence of 5-tuples that contain all the necessary arguments for creating a socket connected to that service. Arguments *af*, *type*, and *proto* (which have the same meaning as for the `socket()` function) can be used to filter which kind of addresses are returned. If a parameter is not specified or zero, all combinations of addresses can be returned (requiring filtering on the user side).

The resulting list of 5-tuples has the following structure:

```
(family, type, proto, canonname, sockaddr)
```

The following example shows how to connect to a given url:

```
s = socket.socket()
# This assumes that if "type" is not specified, an address for
# SOCK_STREAM will be returned, which may be not true
s.connect(socket.getaddrinfo('www.micropython.org', 80)[0][-1])
```

Recommended use of filtering params:

```
s = socket.socket()
# Guaranteed to return an address which can be connect'ed to for
# stream operation.
s.connect(socket.getaddrinfo('www.micropython.org', 80, 0, SOCK_STREAM)[0][-1])
```

**Difference to CPython**

CPython raises a `socket.gaierror` exception (`OSError` subclass) in case of error in this function. MicroPython doesn't have `socket.gaierror` and raises OSError directly. Note that error numbers of `getaddrinfo()` form a separate namespace and may not match error numbers from the `errno` module. To distinguish `getaddrinfo()` errors, they are represented by negative numbers, whereas standard system errors are positive numbers (error numbers are accessible using `e.args[0]` property from an exception object). The use of negative values is a provisional detail which may change in the future.

### `socket.inet_ntop`(*af*, *bin_addr*)

Convert a binary network address *bin_addr* of the given address family *af* to a textual representation:

```
>>> socket.inet_ntop(socket.AF_INET, b"\x7f\0\0\1")
'127.0.0.1'
```

### `socket.inet_pton`(*af*, *txt_addr*)

Convert a textual network address *txt_addr* of the given address family *af* to a binary representation:

```
>>> socket.inet_pton(socket.AF_INET, "1.2.3.4")
b'\x01\x02\x03\x04'
```

## Constants

`socket.AF_INET`

`socket.AF_INET6`

Address family types. Availability depends on a particular MicroPython port.

`socket.SOCK_STREAM`

`socket.SOCK_DGRAM`

Socket types.

`socket.IPPROTO_UDP`

`socket.IPPROTO_TCP`

IP protocol numbers. Availability depends on a particular MicroPython port. Note that you don't need to specify these in a call to `socket.socket()`, because `SOCK_STREAM` socket type automatically selects `IPPROTO_TCP`, and `SOCK_DGRAM` - `IPPROTO_UDP`. Thus, the only real use of these constants is as an argument to `setsockopt()`.

`socket.SOL_*`

Socket option levels (an argument to `setsockopt()`). The exact inventory depends on a MicroPython port.

`socket.SO_*`

Socket options (an argument to `setsockopt()`). The exact inventory depends on a MicroPython port.

Constants specific to WiPy:

`socket.IPPROTO_SEC`

Special protocol value to create SSL-compatible socket.

## class socket

*class* `socket.socket`*(af=AF_INET, type=SOCK_STREAM, proto=IPPROTO_TCP, /)*

Create a new socket using the given address family, socket type and protocol number. Note that specifying *proto* in most cases is not required (and not recommended, as some MicroPython ports may omit `IPPROTO_*` constants). Instead, *type* argument will select needed protocol automatically:

```
# Create STREAM TCP socket
socket(AF_INET, SOCK_STREAM)
# Create DGRAM UDP socket
socket(AF_INET, SOCK_DGRAM)
```

## Methods

**`socket.close()`**

Mark the socket closed and release all resources. Once that happens, all future operations on the socket object will fail. The remote end will receive EOF indication if supported by protocol.

Sockets are automatically closed when they are garbage-collected, but it is recommended to `close()` them explicitly as soon you finished working with them.

**`socket.bind`(*address*)**

Bind the socket to *address*. The socket must not already be bound.

**`socket.listen`( [ *backlog* ] )**

Enable a server to accept connections. If *backlog* is specified, it must be at least 0 (if it's lower, it will be set to 0); and specifies the number of unaccepted connections that the system will allow before refusing new connections. If not specified, a default reasonable value is chosen.

**`socket.accept()`**

Accept a connection. The socket must be bound to an address and listening for connections. The return value is a pair (conn, address) where conn is a new socket object usable to send and receive data on the connection, and address is the address bound to the socket on the other end of the connection.

**`socket.connect`(*address*)**

Connect to a remote socket at *address*.

**`socket.send`(*bytes*)**

Send data to the socket. The socket must be connected to a remote socket. Returns number of bytes sent, which may be smaller than the length of data ("short write").

`socket.sendall`*(bytes)*

Send all data to the socket. The socket must be connected to a remote socket. Unlike `send()`, this method will try to send all of data, by sending data chunk by chunk consecutively.

The behaviour of this method on non-blocking sockets is undefined. Due to this, on MicroPython, it's recommended to use `write()` method instead, which has the same "no short writes" policy for blocking sockets, and will return number of bytes sent on non-blocking sockets.

`socket.recv`*(bufsize)*

Receive data from the socket. The return value is a bytes object representing the data received. The maximum amount of data to be received at once is specified by bufsize.

`socket.sendto`*(bytes, address)*

Send data to the socket. The socket should not be connected to a remote socket, since the destination socket is specified by *address*.

`socket.recvfrom`*(bufsize)*

Receive data from the socket. The return value is a pair *(bytes, address)* where *bytes* is a bytes object representing the data received and *address* is the address of the socket sending the data.

`socket.setsockopt`*(level, optname, value)*

Set the value of the given socket option. The needed symbolic constants are defined in the socket module (SO_* etc.). The *value* can be an integer or a bytes-like object representing a buffer.

`socket.settimeout`*(value)*

**Note**: Not every port supports this method, see below.

Set a timeout on blocking socket operations. The value argument can be a nonnegative floating point number expressing seconds, or None. If a non-zero value is given, subsequent socket operations will raise an `OSError` exception if the timeout period value has elapsed before the operation has completed. If zero is given, the socket is put in non-blocking mode. If None is given, the socket is put in blocking mode.

Not every MicroPython port supports this method. A more portable and generic solution is to use `select.poll` object. This allows to wait on multiple objects at the same time (and not just on sockets, but on generic `stream` objects which support polling). Example:

```
# Instead of:
s.settimeout(1.0)  # time in seconds
s.read(10)  # may timeout

# Use:
poller = select.poll()
poller.register(s, select.POLLIN)
res = poller.poll(1000)  # time in milliseconds
if not res:
    # s is still not ready for input, i.e. operation timed out
```

**Difference to CPython**

CPython raises a `socket.timeout` exception in case of timeout, which is an `OSError` subclass. MicroPython raises an OSError directly instead. If you use `except OSError:` to catch the exception, your code will work both in MicroPython and CPython.

### `socket.setblocking`(*flag*)

Set blocking or non-blocking mode of the socket: if flag is false, the socket is set to non-blocking, else to blocking mode.

This method is a shorthand for certain `settimeout()` calls:

- `sock.setblocking(True)` is equivalent to `sock.settimeout(None)`
- `sock.setblocking(False)` is equivalent to `sock.settimeout(0)`

### `socket.makefile`(*mode='rb', buffering=0, /*)

Return a file object associated with the socket. The exact returned type depends on the arguments given to makefile(). The support is limited to binary modes only (‘rb’, ‘wb’, and ‘rwb’). CPython’s arguments: *encoding*, *errors* and *newline* are not supported.

**Difference to CPython**

As MicroPython doesn’t support buffered streams, values of *buffering* parameter is ignored and treated as if it was 0 (unbuffered).

**Difference to CPython**

Closing the file object returned by makefile() WILL close the original socket as well.

`socket.read(` [ *size* ] `)`

Read up to size bytes from the socket. Return a bytes object. If *size* is not given, it reads all data available from the socket until EOF; as such the method will not return until the socket is closed. This function tries to read as much data as requested (no “short reads”). This may be not possible with non-blocking socket though, and then less data will be returned.

`socket.readinto(`*buf* [ , *nbytes* ] `)`

Read bytes into the *buf*. If *nbytes* is specified then read at most that many bytes. Otherwise, read at most *len(buf)* bytes. Just as `read()`, this method follows “no short reads” policy.

Return value: number of bytes read and stored into *buf*.

`socket.readline()`

Read a line, ending in a newline character.

Return value: the line read.

`socket.write(`*buf*`)`

Write the buffer of bytes to the socket. This function will try to write all data to a socket (no “short writes”). This may be not possible with a non-blocking socket though, and returned value will be less than the length of *buf*.

Return value: number of bytes written.

*exception* `socket.error`

MicroPython does NOT have this exception.

Difference to CPython

CPython used to have a `socket.error` exception which is now deprecated, and is an alias of `OSError`. In MicroPython, use `OSError` directly.