18.1. `socket` — Low-level networking interface

This module provides access to the BSD *socket* interface. It is available on all modern Unix systems, Windows, MacOS, and probably additional platforms.

Note: Some behavior may be platform dependent, since calls are made to the operating system socket APIs.

The Python interface is a straightforward transliteration of the Unix system call and library interface for sockets to Python's object-oriented style: the `socket()` function returns a *socket object* whose methods implement the various socket system calls. Parameter types are somewhat higher-level than in the C interface: as with `read()` and `write()` operations on Python files, buffer allocation on receive operations is automatic, and buffer length is implicit on send operations.

See also:

Module `socketserver`
: Classes that simplify writing network servers.

Module `ssl`
: A TLS/SSL wrapper for socket objects.

18.1.1. Socket families

Depending on the system and the build options, various socket families are supported by this module.

The address format required by a particular socket object is automatically selected based on the address family specified when the socket object was created. Socket addresses are represented as follows:

- The address of an `AF_UNIX` socket bound to a file system node is represented as a string, using the file system encoding and the `'surrogateescape'` error handler (see **PEP 383**). An address in Linux's abstract namespace is returned as a `bytes` object with an initial null byte; note that sockets in this namespace can communicate with normal file system sockets, so programs intended to run on Linux may need to deal with both types of address. A string or `bytes` object can be used for either type of address when passing it as an argument.

Changed in version 3.3: Previously, `AF_UNIX` socket paths were assumed to use UTF-8 encoding.

- A pair `(host, port)` is used for the `AF_INET` address family, where *host* is a string representing either a hostname in Internet domain notation like `'daring.cwi.nl'` or an IPv4 address like `'100.50.200.5'`, and *port* is an integer.
- For `AF_INET6` address family, a four-tuple `(host, port, flowinfo, scopeid)` is used, where *flowinfo* and *scopeid* represent the `sin6_flowinfo` and `sin6_scope_id` members in `struct sockaddr_in6` in C. For `socket` module methods, *flowinfo* and *scopeid* can be omitted just for backward compatibility. Note, however, omission of *scopeid* can cause problems in manipulating scoped IPv6 addresses.
- `AF_NETLINK` sockets are represented as pairs `(pid, groups)`.
- Linux-only support for TIPC is available using the `AF_TIPC` address family. TIPC is an open, non-IP based networked protocol designed for use in clustered computer environments. Addresses are represented by a tuple, and the fields depend on the address type. The general tuple form is `(addr_type, v1, v2, v3 [, scope])`, where:
  - *addr_type* is one of `TIPC_ADDR_NAMESEQ`, `TIPC_ADDR_NAME`, or `TIPC_ADDR_ID`.
  - *scope* is one of `TIPC_ZONE_SCOPE`, `TIPC_CLUSTER_SCOPE`, and `TIPC_NODE_SCOPE`.
  - If *addr_type* is `TIPC_ADDR_NAME`, then *v1* is the server type, *v2* is the port identifier, and *v3* should be 0.

    If *addr_type* is `TIPC_ADDR_NAMESEQ`, then *v1* is the server type, *v2* is the lower port number, and *v3* is the upper port number.

    If *addr_type* is `TIPC_ADDR_ID`, then *v1* is the node, *v2* is the reference, and *v3* should be set to 0.
- A tuple `(interface, )` is used for the `AF_CAN` address family, where *interface* is a string representing a network interface name like `'can0'`. The network interface name `''` can be used to receive packets from all network interfaces of this family.
- A string or a tuple `(id, unit)` is used for the `SYSPROTO_CONTROL` protocol of the `PF_SYSTEM` family. The string is the name of a kernel control using a

dynamically-assigned ID. The tuple can be used if ID and unit number of the kernel control are known or if a registered ID is used.

New in version 3.3.

- Certain other address families (`AF_BLUETOOTH`, `AF_PACKET`, `AF_CAN`) support specific representations.

For IPv4 addresses, two special forms are accepted instead of a host address: the empty string represents `INADDR_ANY`, and the string `'<broadcast>'` represents `INADDR_BROADCAST`. This behavior is not compatible with IPv6, therefore, you may want to avoid these if you intend to support IPv6 with your Python programs.

If you use a hostname in the *host* portion of IPv4/v6 socket address, the program may show a nondeterministic behavior, as Python uses the first address returned from the DNS resolution. The socket address will be resolved differently into an actual IPv4/v6 address, depending on the results from DNS resolution and/or the host configuration. For deterministic behavior use a numeric address in *host* portion.

All errors raise exceptions. The normal exceptions for invalid argument types and out-of-memory conditions can be raised; starting from Python 3.3, errors related to socket or address semantics raise `OSError` or one of its subclasses (they used to raise `socket.error`).

Non-blocking mode is supported through `setblocking()`. A generalization of this based on timeouts is supported through `settimeout()`.

18.1.2. Module contents

The module `socket` exports the following elements.

18.1.2.1. Exceptions

exception `socket.`**`error`**

A deprecated alias of `OSError`.

Changed in version 3.3: Following **PEP 3151**, this class was made an alias of `OSError`.

exception `socket.`**`herror`**

A subclass of `OSError`, this exception is raised for address-related errors, i.e. for functions that use *h_errno* in the POSIX C API, including `gethostbyname_ex()` and `gethostbyaddr()`. The accompanying value is a pair `(h_errno, string)` representing an error returned by a library call.

h_errno is a numeric value, while *string* represents the description of *h_errno*, as returned by the `hstrerror()` C function.

Changed in version 3.3: This class was made a subclass of `OSError`.

exception `socket.`**`gaierror`**

A subclass of `OSError`, this exception is raised for address-related errors by `getaddrinfo()` and `getnameinfo()`. The accompanying value is a pair `(error, string)` representing an error returned by a library call. *string* represents the description of *error*, as returned by the `gai_strerror()` C function. The numeric *error* value will match one of the `EAI_*` constants defined in this module.

Changed in version 3.3: This class was made a subclass of `OSError`.

exception `socket.`**`timeout`**

A subclass of `OSError`, this exception is raised when a timeout occurs on a socket which has had timeouts enabled via a prior call to `settimeout()` (or implicitly through `setdefaulttimeout()`). The accompanying value is a string whose value is currently always “timed out”.

Changed in version 3.3: This class was made a subclass of `OSError`.

18.1.2.2. Constants

The AF_* and SOCK_* constants are now `AddressFamily` and `SocketKind` `IntEnum` collections.

New in version 3.4.

`socket.`**`AF_UNIX`**
`socket.`**`AF_INET`**
`socket.`**`AF_INET6`**

These constants represent the address (and protocol) families, used for the first argument to `socket()`. If the `AF_UNIX` constant is not defined then this protocol is unsupported. More constants may be available depending on the system.

`socket.`**`SOCK_STREAM`**
`socket.`**`SOCK_DGRAM`**
`socket.`**`SOCK_RAW`**
`socket.`**`SOCK_RDM`**
`socket.`**`SOCK_SEQPACKET`**

These constants represent the socket types, used for the second argument to `socket()`. More constants may be available depending on the system. (Only `SOCK_STREAM` and `SOCK_DGRAM` appear to be generally useful.)

`socket.`**`SOCK_CLOEXEC`**
`socket.`**`SOCK_NONBLOCK`**

These two constants, if defined, can be combined with the socket types and allow you to set some flags atomically (thus avoiding possible race conditions and the need for separate calls).

See also: Secure File Descriptor Handling for a more thorough explanation.

Availability: Linux >= 2.6.27.

New in version 3.2.

`SO_*`
`socket.`**`SOMAXCONN`**
`MSG_*`
`SOL_*`
`SCM_*`
`IPPROTO_*`
`IPPORT_*`
`INADDR_*`
`IP_*`
`IPV6_*`
`EAI_*`
`AI_*`
`NI_*`
`TCP_*`

Many constants of these forms, documented in the Unix documentation on sockets and/or the IP protocol, are also defined in the socket module. They are generally used in arguments to the `setsockopt()` and `getsockopt()` methods of socket objects. In most cases, only those symbols that are defined in the Unix header files are defined; for a few symbols, default values are provided.

`socket.`**`AF_CAN`**
`socket.`**`PF_CAN`**
`SOL_CAN_*`
`CAN_*`

Many constants of these forms, documented in the Linux documentation, are also defined in the socket module.

Availability: Linux >= 2.6.25.

New in version 3.3.

`socket.`**`CAN_BCM`**
`CAN_BCM_*`

CAN_BCM, in the CAN protocol family, is the broadcast manager (BCM) protocol. Broadcast manager constants, documented in the Linux documentation, are also defined in the socket module.

Availability: Linux >= 2.6.25.

New in version 3.4.

`socket.`**`AF_RDS`**
`socket.`**`PF_RDS`**
`socket.`**`SOL_RDS`**
`RDS_*`

Many constants of these forms, documented in the Linux documentation, are also defined in the socket module.

Availability: Linux >= 2.6.30.

New in version 3.3.

`SIO_*`
`RCVALL_*`

Constants for Windows' WSAIoctl(). The constants are used as arguments to the `ioctl()` method of socket objects.

`TIPC_*`

TIPC related constants, matching the ones exported by the C socket API. See the TIPC documentation for more information.

`socket.`**`AF_LINK`**

Availability: BSD, OSX.

New in version 3.4.

`socket.`**`has_ipv6`**

This constant contains a boolean value which indicates if IPv6 is supported on this platform.

18.1.2.3. Functions

18.1.2.3.1. Creating sockets

The following functions all create *socket objects*.

`socket.`**`socket`**(*family=AF_INET*, *type=SOCK_STREAM*, *proto=0*, *fileno=None*)

Create a new socket using the given address family, socket type and protocol number. The address family should be `AF_INET` (the default), `AF_INET6`, `AF_UNIX`, `AF_CAN` or `AF_RDS`. The socket type should be `SOCK_STREAM` (the default), `SOCK_DGRAM`, `SOCK_RAW` or perhaps one of the other `SOCK_` constants. The protocol number is usually zero and may be omitted or in the case where the address family is `AF_CAN` the protocol should be one of `CAN_RAW` or `CAN_BCM`.

The newly created socket is *non-inheritable*.

Changed in version 3.3: The AF_CAN family was added. The AF_RDS family was added.

Changed in version 3.4: The CAN_BCM protocol was added.

Changed in version 3.4: The returned socket is now non-inheritable.

`socket.`**`socketpair`**([*family*[, *type*[, *proto*]]])

Build a pair of connected socket objects using the given address family, socket type, and protocol number. Address family, socket type, and protocol number are as for the `socket()` function above. The default family is `AF_UNIX` if defined on the platform; otherwise, the default is `AF_INET`. Availability: Unix.

The newly created sockets are *non-inheritable*.

Changed in version 3.2: The returned socket objects now support the whole socket API, rather than a subset.

Changed in version 3.4: The returned sockets are now non-inheritable.

`socket.`**`create_connection`**(*address*[, *timeout*[, *source_address*]])

Connect to a TCP service listening on the Internet *address* (a 2-tuple `(host, port)`), and return the socket object. This is a higher-level function than `socket.connect()`: if *host* is a non-numeric hostname, it will try to resolve it for both `AF_INET` and `AF_INET6`, and then try to connect to all possible addresses in turn until a connection succeeds. This makes it easy to write clients that are compatible to both IPv4 and IPv6.

Passing the optional *timeout* parameter will set the timeout on the socket instance before attempting to connect. If no *timeout* is supplied, the global default timeout setting returned by `getdefaulttimeout()` is used.

If supplied, *source_address* must be a 2-tuple `(host, port)` for the socket to bind to as its source address before connecting. If host or port are '' or 0 respectively the OS default behavior will be used.

Changed in version 3.2: source_address was added.

Changed in version 3.2: support for the `with` statement was added.

`socket.`**`fromfd`**(*fd*, *family*, *type*, *proto=0*)

Duplicate the file descriptor *fd* (an integer as returned by a file object's `fileno()` method) and build a socket object from the result. Address family, socket type and protocol number are as for the `socket()` function above. The file descriptor should refer to a socket, but this is not checked — subsequent operations on the object may fail if the file descriptor is invalid. This function is rarely needed, but can be used to get or set socket options on a socket passed to a program as standard input or output (such as a server started by the Unix inet daemon). The socket is assumed to be in blocking mode.

The newly created socket is *non-inheritable*.

Changed in version 3.4: The returned socket is now non-inheritable.

`socket.`**`fromshare`**(*data*)

Instantiate a socket from data obtained from the `socket.share()` method. The socket is assumed to be in blocking mode.

Availability: Windows.

New in version 3.3.

`socket.`**`SocketType`**

This is a Python type object that represents the socket object type. It is the same as `type(socket(...))`.

18.1.2.3.2. Other functions

The `socket` module also offers various network-related services:

`socket.`**`getaddrinfo`**(*host*, *port*, *family=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. *host* is a domain name, a string representation of an IPv4/v6 address or `None`. *port* is a string service name such as `'http'`, a numeric port number or `None`. By passing `None` as the value of *host* and *port*, you can pass `NULL` to the underlying C API.

The *family*, *type* and *proto* arguments can be optionally specified in order to narrow the list of addresses returned. Passing zero as a value for each of these arguments selects the full range of results. The *flags* argument can be one or several of the `AI_*` constants, and will influence how results are computed and returned. For example, `AI_NUMERICHOST` will disable domain name resolution and will raise an error if *host* is a domain name.

The function returns a list of 5-tuples with the following structure:

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

In these tuples, *family*, *type*, *proto* are all integers and are meant to be passed to the `socket()` function. *canonname* will be a string representing the canonical name of the *host* if `AI_CANONNAME` is part of the *flags* argument; else *canonname* will be empty. *sockaddr* is a tuple describing a socket address, whose format depends on the returned *family* (a `(address, port)` 2-tuple for `AF_INET`, a `(address, port, flow info, scope id)` 4-tuple for `AF_INET6`), and is meant to be passed to the `socket.connect()` method.

The following example fetches address information for a hypothetical TCP connection to `www.python.org` on port 80 (results may differ on your system if IPv6 isn't enabled):

```
>>> socket.getaddrinfo("www.python.org", 80, proto=socket.IPP
[(2, 1, 6, '', ('82.94.164.162', 80)),
 (10, 1, 6, '', ('2001:888:2000:d::a2', 80, 0, 0))]
```

Changed in version 3.2: parameters can now be passed using keyword arguments.

socket.**getfqdn**([*name*])

Return a fully qualified domain name for *name*. If *name* is omitted or empty, it is interpreted as the local host. To find the fully qualified name, the hostname returned by `gethostbyaddr()` is checked, followed by aliases for the host, if available. The first name which includes a period is selected. In case no fully qualified domain name is available, the hostname as returned by `gethostname()` is returned.

socket.**gethostbyname**(*hostname*)

Translate a host name to IPv4 address format. The IPv4 address is returned as a string, such as `'100.50.200.5'`. If the host name is an IPv4 address itself it is returned unchanged. See `gethostbyname_ex()` for a more complete interface. `gethostbyname()` does not support IPv6 name resolution, and `getaddrinfo()` should be used instead for IPv4/v6 dual stack support.

socket.**gethostbyname_ex**(*hostname*)

Translate a host name to IPv4 address format, extended interface. Return a triple `(hostname, aliaslist, ipaddrlist)` where *hostname* is the primary host name responding to the given *ip_address*, *aliaslist* is a (possibly empty) list of alternative host names for the same address, and *ipaddrlist* is a list of IPv4 addresses for the same interface on the same host (often but not always a single address). `gethostbyname_ex()` does not support IPv6 name resolution, and `getaddrinfo()` should be used instead for IPv4/v6 dual stack support.

socket.**gethostname**()

Return a string containing the hostname of the machine where the Python interpreter is currently executing.

If you want to know the current machine's IP address, you may want to use `gethostbyname(gethostname())`. This operation assumes that there is a valid address-to-host mapping for the host, and the assumption does not always hold.

Note: `gethostname()` doesn't always return the fully qualified domain name; use `getfqdn()` for that.

socket.**gethostbyaddr**(*ip_address*)

Return a triple `(hostname, aliaslist, ipaddrlist)` where *hostname* is the primary host name responding to the given *ip_address*, *aliaslist* is a (possibly empty) list of alternative host names for the same address, and *ipaddrlist* is a list of IPv4/v6 addresses for the same interface on the same host (most likely containing only a single address). To find the fully qualified domain name, use the function `getfqdn()`. `gethostbyaddr()` supports both IPv4 and IPv6.

socket.**getnameinfo**(*sockaddr*, *flags*)

Translate a socket address *sockaddr* into a 2-tuple `(host, port)`. Depending on the settings of *flags*, the result can contain a fully-qualified domain name or numeric address representation in *host*. Similarly, *port* can contain a string port name or a numeric port number.

socket.**getprotobyname**(*protocolname*)

Translate an Internet protocol name (for example, `'icmp'`) to a constant suitable for passing as the (optional) third argument to the `socket()` function.

This is usually only needed for sockets opened in “raw” mode (`SOCK_RAW`); for the normal socket modes, the correct protocol is chosen automatically if the protocol is omitted or zero.

`socket.`**`getservbyname`**(*servicename*[, *protocolname*])

Translate an Internet service name and protocol name to a port number for that service. The optional protocol name, if given, should be `'tcp'` or `'udp'`, otherwise any protocol will match.

`socket.`**`getservbyport`**(*port*[, *protocolname*])

Translate an Internet port number and protocol name to a service name for that service. The optional protocol name, if given, should be `'tcp'` or `'udp'`, otherwise any protocol will match.

`socket.`**`ntohl`**(*x*)

Convert 32-bit positive integers from network to host byte order. On machines where the host byte order is the same as network byte order, this is a no-op; otherwise, it performs a 4-byte swap operation.

`socket.`**`ntohs`**(*x*)

Convert 16-bit positive integers from network to host byte order. On machines where the host byte order is the same as network byte order, this is a no-op; otherwise, it performs a 2-byte swap operation.

`socket.`**`htonl`**(*x*)

Convert 32-bit positive integers from host to network byte order. On machines where the host byte order is the same as network byte order, this is a no-op; otherwise, it performs a 4-byte swap operation.

`socket.`**`htons`**(*x*)

Convert 16-bit positive integers from host to network byte order. On machines where the host byte order is the same as network byte order, this is a no-op; otherwise, it performs a 2-byte swap operation.

`socket.`**`inet_aton`**(*ip_string*)

Convert an IPv4 address from dotted-quad string format (for example, ‘123.45.67.89’) to 32-bit packed binary format, as a bytes object four characters in length. This is useful when conversing with a program that uses the standard C library and needs objects of type `struct in_addr`, which is the C type for the 32-bit packed binary this function returns.

`inet_aton()` also accepts strings with less than three dots; see the Unix manual page *inet(3)* for details.

If the IPv4 address string passed to this function is invalid, `OSError` will be raised. Note that exactly what is valid depends on the underlying C implementation of `inet_aton()`.

`inet_aton()` does not support IPv6, and `inet_pton()` should be used instead for IPv4/v6 dual stack support.

`socket.`**`inet_ntoa`**(*packed_ip*)

Convert a 32-bit packed IPv4 address (a bytes object four characters in length) to its standard dotted-quad string representation (for example, ‘123.45.67.89’). This is useful when conversing with a program that uses the standard C library and needs objects of type `struct in_addr`, which is the C type for the 32-bit packed binary data this function takes as an argument.

If the byte sequence passed to this function is not exactly 4 bytes in length, `OSError` will be raised. `inet_ntoa()` does not support IPv6, and `inet_ntop()` should be used instead for IPv4/v6 dual stack support.

`socket.`**`inet_pton`**(*address_family*, *ip_string*)

Convert an IP address from its family-specific string format to a packed, binary format. `inet_pton()` is useful when a library or network protocol calls for an object of type `struct in_addr` (similar to `inet_aton()`) or `struct in6_addr`.

Supported values for *address_family* are currently `AF_INET` and `AF_INET6`. If the IP address string *ip_string* is invalid, `OSError` will be raised. Note that exactly what is valid depends on both the value of *address_family* and the underlying implementation of `inet_pton()`.

Availability: Unix (maybe not all platforms), Windows.

Changed in version 3.4: Windows support added

`socket.`**`inet_ntop`**(*address_family*, *packed_ip*)

Convert a packed IP address (a bytes object of some number of characters) to its standard, family-specific string representation (for example, `'7.10.0.5'` or `'5aef:2b::8'`). `inet_ntop()` is useful when a library or network protocol returns an object of type `struct in_addr` (similar to `inet_ntoa()`) or `struct in6_addr`.

Supported values for *address_family* are currently `AF_INET` and `AF_INET6`. If the string *packed_ip* is not the correct length for the specified address family, `ValueError` will be raised. A `OSError` is raised for errors from the call to `inet_ntop()`.

Availability: Unix (maybe not all platforms), Windows.

Changed in version 3.4: Windows support added

`socket.`**`CMSG_LEN`**(*length*)

Return the total length, without trailing padding, of an ancillary data item with associated data of the given *length*. This value can often be used as the buffer size for `recvmsg()` to receive a single item of ancillary data, but **RFC 3542** requires portable applications to use `CMSG_SPACE()` and thus include space for padding, even when the item will be the last in the buffer. Raises `OverflowError` if *length* is outside the permissible range of values.

Availability: most Unix platforms, possibly others.

New in version 3.3.

`socket.`**`CMSG_SPACE`**(*length*)

Return the buffer size needed for `recvmsg()` to receive an ancillary data item with associated data of the given *length*, along with any trailing padding. The buffer space needed to receive multiple items is the sum of the `CMSG_SPACE()` values for their associated data lengths. Raises `OverflowError` if *length* is outside the permissible range of values.

Note that some systems might support ancillary data without providing this function. Also note that setting the buffer size using the results of this function may not precisely limit the amount of ancillary data that can be received, since additional data may be able to fit into the padding area.

Availability: most Unix platforms, possibly others.

New in version 3.3.

`socket.`**`getdefaulttimeout`**()

Return the default timeout in seconds (float) for new socket objects. A value of `None` indicates that new socket objects have no timeout. When the socket module is first imported, the default is `None`.

`socket.`**`setdefaulttimeout`**(*timeout*)

Set the default timeout in seconds (float) for new socket objects. When the socket module is first imported, the default is `None`. See `settimeout()` for possible values and their respective meanings.

`socket.`**`sethostname`**(*name*)

Set the machine's hostname to *name*. This will raise a `OSError` if you don't have enough rights.

Availability: Unix.

New in version 3.3.

`socket.`**`if_nameindex`**()

Return a list of network interface information (index int, name string) tuples. `OSError` if the system call fails.

Availability: Unix.

New in version 3.3.

`socket.`**`if_nametoindex`**(*if_name*)

Return a network interface index number corresponding to an interface name. `OSError` if no interface with the given name exists.

Availability: Unix.

New in version 3.3.

`socket.`**`if_indextoname`**(*if_index*)

Return a network interface name corresponding to a interface index number. `OSError` if no interface with the given index exists.

Availability: Unix.

New in version 3.3.

18.1.3. Socket Objects

Socket objects have the following methods. Except for `makefile()`, these correspond to Unix system calls applicable to sockets.

`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.

The newly created socket is *non-inheritable*.

Changed in version 3.4: The socket is now non-inheritable.

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

Bind the socket to *address*. The socket must not already be bound. (The format of *address* depends on the address family — see above.)

socket.**close**()

Mark the socket closed. The underlying system resource (e.g. a file descriptor) is also closed when all file objects from `makefile()` are closed. Once that happens, all future operations on the socket object will fail. The remote end will receive no more data (after queued data is flushed).

Sockets are automatically closed when they are garbage-collected, but it is recommended to `close()` them explicitly, or to use a `with` statement around them.

> **Note:** `close()` releases the resource associated with a connection but does not necessarily close the connection immediately. If you want to close the connection in a timely fashion, call `shutdown()` before `close()`.

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

Connect to a remote socket at *address*. (The format of *address* depends on the address family — see above.)

socket.**connect_ex**(*address*)

Like `connect(address)`, but return an error indicator instead of raising an exception for errors returned by the C-level `connect()` call (other problems, such as "host not found," can still raise exceptions). The error indicator is `0` if the operation succeeded, otherwise the value of the `errno` variable. This is useful to support, for example, asynchronous connects.

socket.**detach**()

Put the socket object into closed state without actually closing the underlying file descriptor. The file descriptor is returned, and can be reused for other purposes.

New in version 3.2.

socket.**dup**()

Duplicate the socket.

The newly created socket is *non-inheritable*.

Changed in version 3.4: The socket is now non-inheritable.

socket.**fileno**()

Return the socket's file descriptor (a small integer). This is useful with `select.select()`.

Under Windows the small integer returned by this method cannot be used where a file descriptor can be used (such as `os.fdopen()`). Unix does not have this limitation.

`socket.`**`get_inheritable`**`()`

Get the *inheritable flag* of the socket's file descriptor or socket's handle: `True` if the socket can be inherited in child processes, `False` if it cannot.

New in version 3.4.

`socket.`**`getpeername`**`()`

Return the remote address to which the socket is connected. This is useful to find out the port number of a remote IPv4/v6 socket, for instance. (The format of the address returned depends on the address family — see above.) On some systems this function is not supported.

`socket.`**`getsockname`**`()`

Return the socket's own address. This is useful to find out the port number of an IPv4/v6 socket, for instance. (The format of the address returned depends on the address family — see above.)

`socket.`**`getsockopt`**(*level*, *optname*[, *buflen*])

Return the value of the given socket option (see the Unix man page *getsockopt (2)*). The needed symbolic constants (`SO_*` etc.) are defined in this module. If *buflen* is absent, an integer option is assumed and its integer value is returned by the function. If *buflen* is present, it specifies the maximum length of the buffer used to receive the option in, and this buffer is returned as a bytes object. It is up to the caller to decode the contents of the buffer (see the optional built-in module `struct` for a way to decode C structures encoded as byte strings).

`socket.`**`gettimeout`**`()`

Return the timeout in seconds (float) associated with socket operations, or `None` if no timeout is set. This reflects the last call to `setblocking()` or `settimeout()`.

`socket.`**`ioctl`**(*control*, *option*)

Platform:	Windows

The `ioctl()` method is a limited interface to the WSAIoctl system interface. Please refer to the Win32 documentation for more information.

On other platforms, the generic `fcntl.fcntl()` and `fcntl.ioctl()` functions may be used; they accept a socket object as their first argument.

socket.**listen**(*backlog*)

Listen for connections made to the socket. The *backlog* argument specifies the maximum number of queued connections and should be at least 0; the maximum value is system-dependent (usually 5), the minimum value is forced to 0.

socket.**makefile**(*mode='r'*, *buffering=None*, *, *encoding=None*, *errors=None*, *newline=None*)

Return a *file object* associated with the socket. The exact returned type depends on the arguments given to `makefile()`. These arguments are interpreted the same way as by the built-in `open()` function.

The socket must be in blocking mode; it can have a timeout, but the file object's internal buffer may end up in a inconsistent state if a timeout occurs.

Closing the file object returned by `makefile()` won't close the original socket unless all other file objects have been closed and `socket.close()` has been called on the socket object.

> **Note:** On Windows, the file-like object created by `makefile()` cannot be used where a file object with a file descriptor is expected, such as the stream arguments of `subprocess.Popen()`.

socket.**recv**(*bufsize*[, *flags*])

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*. See the Unix manual page *recv(2)* for the meaning of the optional argument *flags*; it defaults to zero.

> **Note:** For best match with hardware and network realities, the value of *bufsize* should be a relatively small power of 2, for example, 4096.

socket.**recvfrom**(*bufsize*[, *flags*])

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. See the Unix manual page *recv(2)* for the meaning of the optional argument *flags*; it defaults to zero. (The format of *address* depends on the address family — see above.)

socket.**recvmsg**(*bufsize*[, *ancbufsize*[, *flags*]])

Receive normal data (up to *bufsize* bytes) and ancillary data from the socket. The *ancbufsize* argument sets the size in bytes of the internal buffer used to receive the ancillary data; it defaults to 0, meaning that no ancillary data will be received. Appropriate buffer sizes for ancillary data can be calculated using `CMSG_SPACE()` or `CMSG_LEN()`, and items which do not fit into the buffer might be truncated or discarded. The *flags* argument defaults to 0 and has the same meaning as for `recv()`.

The return value is a 4-tuple: `(data, ancdata, msg_flags, address)`. The *data* item is a `bytes` object holding the non-ancillary data received. The *ancdata* item is a list of zero or more tuples `(cmsg_level, cmsg_type, cmsg_data)` representing the ancillary data (control messages) received: *cmsg_level* and *cmsg_type* are integers specifying the protocol level and protocol-specific type respectively, and *cmsg_data* is a `bytes` object holding the associated data. The *msg_flags* item is the bitwise OR of various flags indicating conditions on the received message; see your system documentation for details. If the receiving socket is unconnected, *address* is the address of the sending socket, if available; otherwise, its value is unspecified.

On some systems, `sendmsg()` and `recvmsg()` can be used to pass file descriptors between processes over an `AF_UNIX` socket. When this facility is used (it is often restricted to `SOCK_STREAM` sockets), `recvmsg()` will return, in its ancillary data, items of the form `(socket.SOL_SOCKET, socket.SCM_RIGHTS, fds)`, where *fds* is a `bytes` object representing the new file descriptors as a binary array of the native C `int` type. If `recvmsg()` raises an exception after the system call returns, it will first attempt to close any file descriptors received via this mechanism.

Some systems do not indicate the truncated length of ancillary data items which have been only partially received. If an item appears to extend beyond the end of the buffer, `recvmsg()` will issue a `RuntimeWarning`, and will return the part of it which is inside the buffer provided it has not been truncated before the start of its associated data.

On systems which support the `SCM_RIGHTS` mechanism, the following function will receive up to *maxfds* file descriptors, returning the message data and a list containing the descriptors (while ignoring unexpected conditions such as unrelated control messages being received). See also `sendmsg()`.

```
import socket, array

def recv_fds(sock, msglen, maxfds):
    fds = array.array("i")   # Array of ints
    msg, ancdata, flags, addr = sock.recvmsg(msglen, socket.C
    for cmsg_level, cmsg_type, cmsg_data in ancdata:
        if (cmsg_level == socket.SOL_SOCKET and cmsg_type ==
```

```
            # Append data, ignoring any truncated integers at
            fds.fromstring(cmsg_data[:len(cmsg_data) - (len(c
    return msg, list(fds)
```

Availability: most Unix platforms, possibly others.

New in version 3.3.

socket.**recvmsg_into**(*buffers*[, *ancbufsize*[, *flags*]])

Receive normal data and ancillary data from the socket, behaving as `recvmsg()` would, but scatter the non-ancillary data into a series of buffers instead of returning a new bytes object. The *buffers* argument must be an iterable of objects that export writable buffers (e.g. `bytearray` objects); these will be filled with successive chunks of the non-ancillary data until it has all been written or there are no more buffers. The operating system may set a limit (`sysconf()` value `SC_IOV_MAX`) on the number of buffers that can be used. The *ancbufsize* and *flags* arguments have the same meaning as for `recvmsg()`.

The return value is a 4-tuple: `(nbytes, ancdata, msg_flags, address)`, where *nbytes* is the total number of bytes of non-ancillary data written into the buffers, and *ancdata*, *msg_flags* and *address* are the same as for `recvmsg()`.

Example:

```
>>> import socket
>>> s1, s2 = socket.socketpair()
>>> b1 = bytearray(b'----')
>>> b2 = bytearray(b'0123456789')
>>> b3 = bytearray(b'--------------')
>>> s1.send(b'Mary had a little lamb')
22
>>> s2.recvmsg_into([b1, memoryview(b2)[2:9], b3])
(22, [], 0, None)
>>> [b1, b2, b3]
[bytearray(b'Mary'), bytearray(b'01 had a 9'), bytearray(b'li
```

Availability: most Unix platforms, possibly others.

New in version 3.3.

socket.**recvfrom_into**(*buffer*[, *nbytes*[, *flags*]])

Receive data from the socket, writing it into *buffer* instead of creating a new bytestring. The return value is a pair `(nbytes, address)` where *nbytes* is the number of bytes received and *address* is the address of the socket sending the data. See the Unix manual page *recv(2)* for the meaning of the optional argument *flags*; it defaults to zero. (The format of *address* depends on the address family — see above.)

socket.**recv_into**(*buffer*[, *nbytes*[, *flags*]])

Receive up to *nbytes* bytes from the socket, storing the data into a buffer rather than creating a new bytestring. If *nbytes* is not specified (or 0), receive up to the size available in the given buffer. Returns the number of bytes received. See the Unix manual page *recv(2)* for the meaning of the optional argument *flags*; it defaults to zero.

socket.**send**(*bytes*[, *flags*])

Send data to the socket. The socket must be connected to a remote socket. The optional *flags* argument has the same meaning as for `recv()` above. Returns the number of bytes sent. Applications are responsible for checking that all data has been sent; if only some of the data was transmitted, the application needs to attempt delivery of the remaining data. For further information on this topic, consult the *Socket Programming HOWTO*.

socket.**sendall**(*bytes*[, *flags*])

Send data to the socket. The socket must be connected to a remote socket. The optional *flags* argument has the same meaning as for `recv()` above. Unlike `send()`, this method continues to send data from *bytes* until either all data has been sent or an error occurs. `None` is returned on success. On error, an exception is raised, and there is no way to determine how much data, if any, was successfully sent.

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

Send data to the socket. The socket should not be connected to a remote socket, since the destination socket is specified by *address*. The optional *flags* argument has the same meaning as for `recv()` above. Return the number of bytes sent. (The format of *address* depends on the address family — see above.)

socket.**sendmsg**(*buffers*[, *ancdata*[, *flags*[, *address*]]])

Send normal and ancillary data to the socket, gathering the non-ancillary data from a series of buffers and concatenating it into a single message. The *buffers* argument specifies the non-ancillary data as an iterable of *bytes-like objects* (e.g. `bytes` objects); the operating system may set a limit (`sysconf()` value `SC_IOV_MAX`) on the number of buffers that can be used. The *ancdata* argument specifies the ancillary data (control messages) as an iterable of zero or more tuples `(cmsg_level, cmsg_type, cmsg_data)`, where *cmsg_level* and *cmsg_type* are integers specifying the protocol level and protocol-specific type respectively, and *cmsg_data* is a bytes-like object holding the associated data. Note that some systems (in particular, systems without `CMSG_SPACE()`) might support sending only one control message per call. The *flags* argument defaults to 0 and has the same meaning as for `send()`. If *address* is supplied and not `None`, it sets a destination address for the message. The return value is the number of bytes of non-ancillary data sent.

The following function sends the list of file descriptors *fds* over an `AF_UNIX` socket, on systems which support the `SCM_RIGHTS` mechanism. See also `recvmsg()`.

```
import socket, array

def send_fds(sock, msg, fds):
    return sock.sendmsg([msg], [(socket.SOL_SOCKET, socket.SC
```

Availability: most Unix platforms, possibly others.

New in version 3.3.

`socket.`**`set_inheritable`**(*inheritable*)

Set the *inheritable flag* of the socket's file descriptor or socket's handle.

New in version 3.4.

`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(0.0)`

`socket.`**`settimeout`**(*value*)

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 a `timeout` 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.

For further information, please consult the *notes on socket timeouts*.

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

Set the value of the given socket option (see the Unix manual page *setsockopt (2)*). The needed symbolic constants are defined in the `socket` module (`SO_*` etc.). The value can be an integer or a bytes object representing a buffer. In the latter case it is up to the caller to ensure that the bytestring contains the proper bits (see the optional built-in module `struct` for a way to encode C structures as bytestrings).

`socket.`**`shutdown`**(*how*)

Shut down one or both halves of the connection. If *how* is `SHUT_RD`, further receives are disallowed. If *how* is `SHUT_WR`, further sends are disallowed. If *how* is `SHUT_RDWR`, further sends and receives are disallowed.

`socket.`**`share`**(*process_id*)

Duplicate a socket and prepare it for sharing with a target process. The target process must be provided with *process_id*. The resulting bytes object can then be passed to the target process using some form of interprocess communication and the socket can be recreated there using `fromshare()`. Once this method has been called, it is safe to close the socket since the operating system has already duplicated it for the target process.

Availability: Windows.

New in version 3.3.

Note that there are no methods `read()` or `write()`; use `recv()` and `send()` without *flags* argument instead.

Socket objects also have these (read-only) attributes that correspond to the values given to the `socket` constructor.

`socket.`**`family`**

The socket family.

`socket.`**`type`**

The socket type.

`socket.`**`proto`**

The socket protocol.

18.1.4. Notes on socket timeouts

A socket object can be in one of three modes: blocking, non-blocking, or timeout. Sockets are by default always created in blocking mode, but this can be changed by calling `setdefaulttimeout()`.

- In *blocking mode*, operations block until complete or the system returns an error (such as connection timed out).
- In *non-blocking mode*, operations fail (with an error that is unfortunately system-dependent) if they cannot be completed immediately: functions from the `select` can be used to know when and whether a socket is available for reading or writing.
- In *timeout mode*, operations fail if they cannot be completed within the timeout specified for the socket (they raise a `timeout` exception) or if the system returns an error.

> **Note:** At the operating system level, sockets in *timeout mode* are internally set in non-blocking mode. Also, the blocking and timeout modes are shared between file descriptors and socket objects that refer to the same network endpoint. This implementation detail can have visible consequences if e.g. you decide to use the `fileno()` of a socket.

18.1.4.1. Timeouts and the `connect` method

The `connect()` operation is also subject to the timeout setting, and in general it is recommended to call `settimeout()` before calling `connect()` or pass a timeout parameter to `create_connection()`. However, the system network stack may also return a connection timeout error of its own regardless of any Python socket timeout setting.

18.1.4.2. Timeouts and the `accept` method

If `getdefaulttimeout()` is not `None`, sockets returned by the `accept()` method inherit that timeout. Otherwise, the behaviour depends on settings of the listening socket:

- if the listening socket is in *blocking mode* or in *timeout mode*, the socket returned by `accept()` is in *blocking mode*;
- if the listening socket is in *non-blocking mode*, whether the socket returned by `accept()` is in blocking or non-blocking mode is operating system-dependent. If you want to ensure cross-platform behaviour, it is recommended you manually override this setting.

18.1.5. Example

Here are four minimal example programs using the TCP/IP protocol: a server that echoes all data that it receives back (servicing only one client), and a client using it. Note that a server must perform the sequence `socket()`, `bind()`, `listen()`, `accept()` (possibly repeating the `accept()` to service more than one client), while a client only needs the sequence `socket()`, `connect()`. Also note that the server does not `sendall()`/`recv()` on the socket it is listening on but on the new socket returned by `accept()`.

The first two examples support IPv4 only.

```
# Echo server program
import socket

HOST = ''                 # Symbolic name meaning all available
PORT = 50007              # Arbitrary non-privileged port
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((HOST, PORT))
s.listen(1)
conn, addr = s.accept()
print('Connected by', addr)
while True:
    data = conn.recv(1024)
    if not data: break
    conn.sendall(data)
conn.close()
```

```
# Echo client program
import socket

HOST = 'daring.cwi.nl'    # The remote host
PORT = 50007              # The same port as used by the server
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((HOST, PORT))
s.sendall(b'Hello, world')
data = s.recv(1024)
s.close()
print('Received', repr(data))
```

The next two examples are identical to the above two, but support both IPv4 and IPv6. The server side will listen to the first address family available (it should listen to both instead). On most of IPv6-ready systems, IPv6 will take precedence and the server may not accept IPv4 traffic. The client side will try to connect to the all addresses returned as a result of the name resolution, and sends traffic to the first one connected successfully.

```
# Echo server program
import socket
import sys

HOST = None               # Symbolic name meaning all available
PORT = 50007              # Arbitrary non-privileged port
```

```
s = None
for res in socket.getaddrinfo(HOST, PORT, socket.AF_UNSPEC,
                              socket.SOCK_STREAM, 0, socket.AI_P
    af, socktype, proto, canonname, sa = res
    try:
        s = socket.socket(af, socktype, proto)
    except OSError as msg:
        s = None
        continue
    try:
        s.bind(sa)
        s.listen(1)
    except OSError as msg:
        s.close()
        s = None
        continue
    break
if s is None:
    print('could not open socket')
    sys.exit(1)
conn, addr = s.accept()
print('Connected by', addr)
while True:
    data = conn.recv(1024)
    if not data: break
    conn.send(data)
conn.close()
```

```
# Echo client program
import socket
import sys

HOST = 'daring.cwi.nl'    # The remote host
PORT = 50007              # The same port as used by the server
s = None
for res in socket.getaddrinfo(HOST, PORT, socket.AF_UNSPEC, sock
    af, socktype, proto, canonname, sa = res
    try:
        s = socket.socket(af, socktype, proto)
    except OSError as msg:
        s = None
        continue
    try:
        s.connect(sa)
    except OSError as msg:
        s.close()
        s = None
        continue
    break
if s is None:
    print('could not open socket')
```

```
    sys.exit(1)
s.sendall(b'Hello, world')
data = s.recv(1024)
s.close()
print('Received', repr(data))
```

The next example shows how to write a very simple network sniffer with raw sockets on Windows. The example requires administrator privileges to modify the interface:

```
import socket

# the public network interface
HOST = socket.gethostbyname(socket.gethostname())

# create a raw socket and bind it to the public interface
s = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROT
s.bind((HOST, 0))

# Include IP headers
s.setsockopt(socket.IPPROTO_IP, socket.IP_HDRINCL, 1)

# receive all packages
s.ioctl(socket.SIO_RCVALL, socket.RCVALL_ON)

# receive a package
print(s.recvfrom(65565))

# disabled promiscuous mode
s.ioctl(socket.SIO_RCVALL, socket.RCVALL_OFF)
```

The last example shows how to use the socket interface to communicate to a CAN network using the raw socket protocol. To use CAN with the broadcast manager protocol instead, open a socket with:

```
socket.socket(socket.AF_CAN, socket.SOCK_DGRAM, socket.CAN_BCM)
```

After binding (`CAN_RAW`) or connecting (`CAN_BCM`) the socket, you can use the `socket.send()`, and the `socket.recv()` operations (and their counterparts) on the socket object as usual.

This example might require special priviledge:

```
import socket
import struct


# CAN frame packing/unpacking (see 'struct can_frame' in <linux/
```

```
can_frame_fmt = "=IB3x8s"
can_frame_size = struct.calcsize(can_frame_fmt)

def build_can_frame(can_id, data):
    can_dlc = len(data)
    data = data.ljust(8, b'\x00')
    return struct.pack(can_frame_fmt, can_id, can_dlc, data)

def dissect_can_frame(frame):
    can_id, can_dlc, data = struct.unpack(can_frame_fmt, frame)
    return (can_id, can_dlc, data[:can_dlc])


# create a raw socket and bind it to the 'vcan0' interface
s = socket.socket(socket.AF_CAN, socket.SOCK_RAW, socket.CAN_RAW
s.bind(('vcan0',))

while True:
    cf, addr = s.recvfrom(can_frame_size)

    print('Received: can_id=%x, can_dlc=%x, data=%s' % dissect_c

    try:
        s.send(cf)
    except OSError:
        print('Error sending CAN frame')

    try:
        s.send(build_can_frame(0x01, b'\x01\x02\x03'))
    except OSError:
        print('Error sending CAN frame')
```

Running an example several times with too small delay between executions, could lead to this error:

```
OSError: [Errno 98] Address already in use
```

This is because the previous execution has left the socket in a `TIME_WAIT` state, and can't be immediately reused.

There is a `socket` flag to set, in order to prevent this, `socket.SO_REUSEADDR`:

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s.bind((HOST, PORT))
```

the `SO_REUSEADDR` flag tells the kernel to reuse a local socket in `TIME_WAIT` state, without waiting for its natural timeout to expire.

See also: For an introduction to socket programming (in C), see the following papers:

- *An Introductory 4.3BSD Interprocess Communication Tutorial*, by Stuart Sechrest
- *An Advanced 4.3BSD Interprocess Communication Tutorial*, by Samuel J. Leffler et al,

both in the UNIX Programmer's Manual, Supplementary Documents 1 (sections PS1:7 and PS1:8). The platform-specific reference material for the various socket-related system calls are also a valuable source of information on the details of socket semantics. For Unix, refer to the manual pages; for Windows, see the WinSock (or Winsock 2) specification. For IPv6-ready APIs, readers may want to refer to **RFC 3493** titled Basic Socket Interface Extensions for IPv6.