Socket – Low-level networking interface

This module provides access to the BSD socket interface.

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.

Socket families

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-like 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-like object can be used for either type of address when passing it as an argument.
- 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, scope_id) is used, where flowinfo and scope_id represent the sin6_flowinfo and sin6_scope_id members in struct sockaddr_in6 in C. For socket module methods, flowinfo and scope_id can be omitted just for backward compatibility. Note, however, omission of scope_id 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])
- 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.

AF_ALG is a Linux-only socket based interface to Kernel cryptography. An algorithm socket is configured with a tuple of two to four elements (type, name [, feat [, mask]]), where:

- type is the algorithm type as string, e.g. aead, hash, skcipher or rng.
- name is the algorithm name and operation mode as string, e.g. sha256, hmac(sha256), cbc(aes) or drbg_nopr_ctr_aes256.
- feat and mask are unsigned 32bit integers.

AF_VSOCK allows communication between virtual machines and their hosts. The sockets are represented as a (CID, port) tuple where the context ID or CID and port are integers.

- AF_PACKET is a low-level interface directly to network devices. The packets are represented by the tuple (ifname, proto[, pkttype[, hatype[, addr]]]) where:
    - ifname - String specifying the device name.
    - proto - An in network-byte-order integer specifying the Ethernet protocol number.
    - pkttype - Optional integer specifying the packet type:
        - PACKET_HOST (the default) - Packet addressed to the local host.
        - PACKET_BROADCAST - Physical-layer broadcast packet.
        - PACKET_MULTIHOST - Packet sent to a physical-layer multicast address.
        - PACKET_OTHERHOST - Packet to some other host that has been caught by a device driver in promiscuous mode.
        - PACKET_OUTGOING - Packet originating from the local host that is looped back to a packet socket.
    - hatype - Optional integer specifying the ARP hardware address type.
    - addr - Optional bytes-like object specifying the hardware physical address, whose interpretation depends on the device.

AF_QIPCRTR is a Linux-only socket based interface for communicating with services running on co-processors in Qualcomm platforms. The address family is represented as a (node, port) tuple where the node and port are non-negative integers.

Exceptions

exception socket.error

A deprecated 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.

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.

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”.

Constants

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

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).

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.

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.

socket.CAN_RAW_FD_FRAMES

Enables CAN FD support in a CAN_RAW socket. This is disabled by default. This allows your application to send both CAN and CAN FD frames; however, you must accept both CAN and CAN FD frames when reading from the socket.

socket.CAN_RAW_JOIN_FILTERS

Joins the applied CAN filters such that only CAN frames that match all given CAN filters are passed to user space.

socket.CAN_ISOTP

CAN_ISOTP, in the CAN protocol family, is the ISO-TP (ISO 15765-2) protocol. ISO-TP constants, documented in the Linux documentation.

socket.CAN_J1939

CAN_J1939, in the CAN protocol family, is the SAE J1939 protocol. J1939 constants, documented in the Linux documentation.

socket.AF_PACKET

socket.PF_PACKET

PACKET_*

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

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.

socket.SIO_RCVALL

socket.SIO_KEEPALIVE_VALS

socket.SIO_LOOPBACK_FAST_PATH

RCVALL_*

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

Functions

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, AF_PACKET, 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, CAN_BCM, CAN_ISOTP or CAN_J1939.

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM | socket.SOCK_NONBLOCK)

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.

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.

socket.create_server(address, *, family=AF_INET, backlog=None, reuse_port=False, dualstack_ipv6=False)

import socketaddr = ("", 8080) # all interfaces, port 8080if socket.has_dualstack_ipv6(): s = socket.create_server(addr, family=socket.AF_INET6, dualstack_ipv6=True)else: s = socket.create_server(addr)

socket.has_dualstack_ipv6()

Return True if the platform supports creating a TCP socket which can handle both IPv4 and IPv6 connections.

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.

socket.fromshare(data)

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

Socket.SocketType

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

Socket Objects

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.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.

socket.connect(address)

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

If the connection is interrupted by a signal, the method waits until the connection completes, or raise a socket.timeout on timeout, if the signal handler does not raise an exception and the socket is blocking or has a timeout. For non-blocking sockets, the method raises an InterruptedError exception if the connection is interrupted by a signal (or the exception raised by the signal handler).

Raises an auditing event socket.connect with arguments self, address.

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.

socket.dup()

Duplicate the socket.

The newly created socket is non-inheritable.

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.

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.

Supports IPv4.

 # Echo server programimport socketHOST = '' # Symbolic name meaning all available interfacesPORT = 50007 # Arbitrary non-privileged portwith socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s: s.bind((HOST, PORT)) s.listen(1) conn, addr = s.accept() with conn: print('Connected by', addr) while True: data = conn.recv(1024) if not data: break conn.sendall(data)

Supports IPv4 and IPv6.

Echo server program**import socketimport sysHOST = None # Symbolic name meaning all available interfacesPORT = 50007 # Arbitrary non-privileged ports = Nonefor res in socket.getaddrinfo(HOST, PORT, socket.AF_UNSPEC, socket.SOCK_STREAM, 0, socket.AI_PASSIVE): 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 breakif s is None: print('could not open socket') sys.exit(1)conn, addr = s.accept()with conn: print('Connected by', addr) while True: data = conn.recv(1024) if not data: break conn.send(data)**