`socket` — Low-level networking interface

This module provides access to the BSD *socket* interface. It is available on all modern Unix systems, Windows, MacOS, OS/2, and probably additional platforms.

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

For an introduction to socket programming (in C), see the following papers: An Introductory 4.3BSD Interprocess Communication Tutorial, by Stuart Sechrest and 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.

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.

Socket addresses are represented as follows: A single string is used for the `AF_UNIX` address family. 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 integral port number. For `AF_INET6` address family, a four-tuple `(host, port, flowinfo, scopeid)` is used, where *flowinfo* and *scopeid* represents `sin6_flowinfo` and `sin6_scope_id` member 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. Other address families are currently not supported. The address format required by a particular socket object is automatically selected based on the address family specified when the socket object was created.

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`**. The behavior is not available for IPv6 for backward compatibility,

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.

AF_NETLINK sockets are represented as pairs `pid, groups`.

Linux-only support for TIPC is also 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.

All errors raise exceptions. The normal exceptions for invalid argument types and out-of-memory conditions can be raised; errors related to socket or address semantics raise the error **`socket.error`**.

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

The module **`socket`** exports the following constants and functions:

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

This exception is raised for socket-related errors. The accompanying value is either a string telling what went wrong or a pair `(errno, string)` representing an error returned by a system call, similar to the value accompanying **`os.error`**. See the

module `errno`, which contains names for the error codes defined by the underlying operating system.

exception `socket.`**herror**

This exception is raised for address-related errors, i.e. for functions that use *h_errno* in the C API, including `gethostbyname_ex()` and `gethostbyaddr()`.

The accompanying value is a pair `(h_errno, string)` representing an error returned by a library call. *string* represents the description of *h_errno*, as returned by the **`hstrerror`** C function.

exception `socket.`**gaierror**

This exception is raised for address-related errors, for `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 *error* value will match one of the **`EAI_*`** constants defined in this module.

exception `socket.`**timeout**

This exception is raised when a timeout occurs on a socket which has had timeouts enabled via a prior call to **`settimeout()`**. The accompanying value is a string whose value is currently always “timed out”.

`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.

`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()`. (Only `SOCK_STREAM` and `SOCK_DGRAM` appear to be generally useful.)

SO_*
`socket.`**SOMAXCONN**
MSG_*
SOL_*

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.

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.`**has_ipv6**

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

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

Convenience function. Connect to *address* (a 2-tuple `(host, port)`), and return the socket object. 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.

`socket.`**getaddrinfo**(*host*, *port*[, *family*[, *socktype*[, *proto*[, *flags*]]]])

Resolves the *host*/*port* argument, into a sequence of 5-tuples that contain all the necessary arguments for creating the corresponding socket. *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`**. The rest of the arguments are optional and must be numeric if specified. By passing **`None`** as the value of *host* and *port*, , you can pass `NULL` to the C API.

The `getaddrinfo()` function returns a list of 5-tuples with the following structure:

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

family, *socktype*, *proto* are all integers and are meant to be passed to the `socket()` function. *canonname* is a string representing the canonical name of the *host*. It can be a numeric IPv4/v6 address when **`AI_CANONNAME`** is specified for a numeric *host*. *sockaddr* is a tuple describing a socket address, as described above. See the source for **`socket`** and other library modules for a typical usage of the function.

`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()` (see above).

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

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` or `AF_UNIX`. The socket type should be `SOCK_STREAM` (the default), `SOCK_DGRAM` or perhaps one of the other `SOCK_` constants. The protocol number is usually zero and may be omitted in that case.

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.

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

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. Availability: Unix.

`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.

If the IPv4 address string passed to this function is invalid, **`socket.error`** 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 **`getnameinfo()`** 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, **socket.error** will be raised. **inet_ntoa()** does not support IPv6, and **getnameinfo()** 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, **socket.error** 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).

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 **socket.error** is raised for errors from the call to **inet_ntop()**.

Availability: Unix (maybe not all platforms).

socket.**getdefaulttimeout**()

Return the default timeout in floating seconds 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 floating seconds 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.**SocketType**

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

See also:

Module `socketserver`

Classes that simplify writing network servers.

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.

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**()

Close the socket. 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.

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

Platform: Windows

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

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 1; the maximum value is system-dependent (usually 5).

socket.**makefile**([*mode*[, *bufsize*]])

Return a *file object* associated with the socket. (File objects are described in *File Objects*.) The file object references a **dup**ped version of the socket file descriptor, so the file object and socket object may be closed or garbage-collected independently. The socket must be in blocking mode (it can not have a timeout). The optional *mode* and *bufsize* arguments are interpreted the same way as by the built-in **file()** function.

`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.`**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. 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.

`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*[, *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.`**`setblocking`**(*flag*)

Set blocking or non-blocking mode of the socket: if *flag* is 0, the socket is set to non-blocking, else to blocking mode. Initially all sockets are in blocking mode. In non-blocking mode, if a `recv()` call doesn't find any data, or if a `send()` call can't immediately dispose of the data, a `error` exception is raised; in blocking mode, the calls block until they can proceed. `s.setblocking(0)` is equivalent to `s.settimeout(0)`; `s.setblocking(1)` is equivalent to `s.settimeout(None)`.

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

Set a timeout on blocking socket operations. The *value* argument can be a nonnegative float expressing seconds, or **`None`**. If a float is given, subsequent socket operations will raise an `timeout` exception if the timeout period *value* has elapsed before the operation has completed. Setting a timeout of **`None`** disables timeouts on socket operations. `s.settimeout(0.0)` is equivalent to `s.setblocking(0)`; `s.settimeout(None)` is equivalent to `s.setblocking(1)`.

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

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

Some notes on socket blocking and timeouts: A socket object can be in one of three modes: blocking, non-blocking, or timeout. Sockets are always created in blocking mode. In blocking mode, operations block until complete. In non-blocking mode, operations fail (with an error that is unfortunately system-dependent) if they cannot be completed immediately. In timeout mode, operations fail if they cannot be completed within the

timeout specified for the socket. The **`setblocking()`** method is simply a shorthand for certain **`settimeout()`** calls.

Timeout mode internally sets the socket in non-blocking mode. The blocking and timeout modes are shared between file descriptors and socket objects that refer to the same network endpoint. A consequence of this is that file objects returned by the **`makefile()`** method must only be used when the socket is in blocking mode; in timeout or non-blocking mode file operations that cannot be completed immediately will fail.

Note that the **`connect()`** operation is subject to the timeout setting, and in general it is recommended to call **`settimeout()`** before calling **`connect()`**.

`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.

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.

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 **send()**/**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 interfa
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.send(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.send(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 interfa
PORT = 50007              # Arbitrary non-privileged port
s = None
for 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 socket.error as msg:
        s = None
        continue
    try:
        s.bind(sa)
        s.listen(1)
    except socket.error 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, socket.SOCK
    af, socktype, proto, canonname, sa = res
    try:
        s = socket.socket(af, socktype, proto)
    except socket.error as msg:
        s = None
        continue
    try:
        s.connect(sa)
    except socket.error as msg:
```

```
        s.close()
        s = None
        continue
    break
if s is None:
    print('could not open socket')
    sys.exit(1)
s.send(b'Hello, world')
data = s.recv(1024)
s.close()
print('Received', repr(data))
```

The last 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.IPPROTO_IP)
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)
```