7.2 socket -- Low-level networking interface

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

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.

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 a single string for the `AF_UNIX` address family and as a pair `(`*host*`, `*port*`)` 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 IP address like `'100.50.200.5'`, and *port* is an integral port number. 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 IP addresses, two special forms are accepted instead of a host address: the empty string represents `INADDR_ANY`, and the string `'<broadcast>'` represents `INADDR_BROADCAST`.

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 the `setblocking()` method.

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

`error`
: This exception is raised for socket- or address-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.

`AF_UNIX`
`AF_INET`
: 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.

`SOCK_STREAM`
`SOCK_DGRAM`
`SOCK_RAW`
`SOCK_RDM`
`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_*`
`SOMAXCONN`
`MSG_*`
`SOL_*`
`IPPROTO_*`
`IPPORT_*`
`INADDR_*`
`IP_*`

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.

`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, then 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 is returned. New in version 2.0.

`gethostbyname` (*hostname*)

Translate a host name to IP address format. The IP address is returned as a string, e.g., `'100.50.200.5'`. If the host name is an IP address itself it is returned unchanged. See `gethostbyname_ex()` for a more complete interface.

`gethostbyname_ex` (*hostname*)

Translate a host name to IP 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 IP addresses for the same interface on the same host (often but not always a single address).

`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, use `gethostbyname(gethostname())`. Note: `gethostname()` doesn't always return the fully qualified domain name; use `gethostbyaddr(gethostname())` (see below).

`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 IP 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()`.

`getprotobyname` (*protocolname*)

Translate an Internet protocol name (e.g. `'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.

`getservbyname` (*servicename, protocolname*)

Translate an Internet service name and protocol name to a port number for that service. The protocol name should be `'tcp'` or `'udp'`.

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

`fromfd` (*fd, family, type*[, *proto*])
Build a socket object from an existing file descriptor (an integer as returned by a file object's `fileno()` method). 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 (e.g. a server started by the Unix inet daemon).

`ntohl` (*x*)
Convert 32-bit 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.

`ntohs` (*x*)
Convert 16-bit 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.

`htonl` (*x*)
Convert 32-bit 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.

`htons` (*x*)
Convert 16-bit 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.

`inet_aton` (*ip_string*)
Convert an IP address from dotted-quad string format (e.g. '123.45.67.89') to 32-bit packed binary format, as a string four characters in length.

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 IP 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_ntoa` (*packed_ip*)
Convert a 32-bit packed IP address (a string four characters in length) to its standard dotted-quad string representation (e.g. '123.45.67.89').

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 takes as an argument.

If the string passed to this function is not exactly 4 bytes in length, `socket.error` will be raised.

`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.

Subsections

Previous: 7.1.1 Example **Up:** 7. Optional Operating System **Next:** 7.2.1 Socket Objects

See About this document... for information on suggesting changes.