**What is Sockets?**

Sockets are the endpoints of a bidirectional communications channel. Sockets may communicate within a process, between processes on the same machine, or between processes on different continents.

Sockets may be implemented over a number of different channel types: Unix domain sockets, TCP, UDP, and so on. The *socket* library provides specific classes for handling the common transports as well as a generic interface for handling the rest.

Sockets have their own vocabulary:

|  |  |
| --- | --- |
| **Term** | **Description** |
| domain | The family of protocols that is used as the transport mechanism. These values are constants such as AF\_INET, PF\_INET, PF\_UNIX, PF\_X25, and so on. |
| type | The type of communications between the two endpoints, typically SOCK\_STREAM for connection-oriented protocols and SOCK\_DGRAM for connectionless protocols. |
| protocol | Typically zero, this may be used to identify a variant of a protocol within a domain and type. |
| hostname | The identifier of a network interface:   * A string, which can be a host name, a dotted-quad address, or an IPV6 address in colon (and possibly dot) notation * A string "<broadcast>", which specifies an INADDR\_BROADCAST address. * A zero-length string, which specifies INADDR\_ANY, or * An Integer, interpreted as a binary address in host byte order. |
| port | Each server listens for clients calling on one or more ports. A port may be a Fixnum port number, a string containing a port number, or the name of a service. |

**The *socket* Module**

To create a socket, you must use the *socket.socket()* function available in *socket* module, which has the general syntax −

s = socket.socket (socket\_family, socket\_type, protocol=0)

Here is the description of the parameters −

* **socket\_family:** This is either AF\_UNIX or AF\_INET, as explained earlier.
* **socket\_type:** This is either SOCK\_STREAM or SOCK\_DGRAM.
* **protocol:** This is usually left out, defaulting to 0.

Once you have *socket* object, then you can use required functions to create your client or server program. Following is the list of functions required −

**Server Socket Methods**

|  |  |
| --- | --- |
| **Method** | **Description** |
| s.bind() | This method binds address (hostname, port number pair) to socket. |
| s.listen() | This method sets up and start TCP listener. |
| s.accept() | This passively accept TCP client connection, waiting until connection arrives (blocking). |

**Client Socket Methods**

|  |  |
| --- | --- |
| **Method** | **Description** |
| s.connect() | This method actively initiates TCP server connection. |

**General Socket Methods**

|  |  |
| --- | --- |
| **Method** | **Description** |
| s.recv() | This method receives TCP message |
| s.send() | This method transmits TCP message |
| s.recvfrom() | This method receives UDP message |
| s.sendto() | This method transmits UDP message |
| s.close() | This method closes socket |
| socket.gethostname() | Returns the hostname. |

## 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()](https://docs.python.org/3.0/library/select.html#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()](https://docs.python.org/3.0/library/os.html#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](https://docs.python.org/3.0/library/struct.html#module-struct) for a way to decode C structures encoded as byte strings).

socket.ioctl(control, option)

|  |  |
| --- | --- |
| **Platform:** | Windows |

The [ioctl()](https://docs.python.org/3.0/library/socket.html#socket.socket.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](https://docs.python.org/3.0/library/stdtypes.html#bltin-file-objects).) The file object references a dupped 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()](https://docs.python.org/3.0/library/socket.html#socket.socket.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()](https://docs.python.org/3.0/library/socket.html#socket.socket.recv) above. Unlike [send()](https://docs.python.org/3.0/library/socket.html#socket.socket.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()](https://docs.python.org/3.0/library/socket.html#socket.socket.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()](https://docs.python.org/3.0/library/socket.html#socket.socket.recv) call doesn’t find any data, or if a [send()](https://docs.python.org/3.0/library/socket.html#socket.socket.send) call can’t immediately dispose of the data, a [error](https://docs.python.org/3.0/library/socket.html#socket.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](https://docs.python.org/3.0/library/socket.html#socket.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()](https://docs.python.org/3.0/library/socket.html#socket.socket.setblocking) or [settimeout()](https://docs.python.org/3.0/library/socket.html#socket.socket.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](https://docs.python.org/3.0/library/struct.html#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](https://docs.python.org/3.0/library/socket.html#socket.socket) constructor.

socket.family

The socket family.

socket.type

The socket type.

socket.proto

The socket protocol.