17. Networking Basics - Python in a Nutshell, 3rd Edition

safaribooksonline.com/library/view/python-in-a/9781491913833/ch17.html

Socket creation functions

Socket objects represent network endpoints. There are a number of different functions supplied by the `socket` module to create a socket:

create_connection

`create_connection([address, [timeout, [source_address]]])`

Creates a socket connected to a TCP endpoint at an address (a `(host, port)` pair). `host` can either be a numeric network address or a DNS hostname; in the latter case, name resolution is attempted for both `AF_INET` and `AF_INET6`, and then a connection is attempted to each returned address in turn—a convenient way to create client programs using either IPv6 or IPv4 as appropriate.

The `timeout` argument, if given, specifies the connection timeout in seconds and thereby sets the socket's mode; when not present, the `socket.getdefaulttimeout` function is called to determine the value. The `source_address` argument, if given, must also be a pair `(host, port)` that the remote socket gets passed as the connecting endpoint. When `host` is `''` or `port` is `0`, the default OS behavior is used.

socket

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

Creates and returns a socket of the appropriate address family and type (by default, a TCP socket on IPv4). The protocol number `proto` is only used with CAN sockets. When you pass the `fileno` argument, other arguments are ignored: the function returns the socket already associated with the given file descriptor.

The socket does not get inherited by child processes.

socketpair

`socketpair([family[, type[, proto]]])`

Returns a connected pair of sockets of the given address family, socket type, and (CAN sockets only) protocol. When `family` is not specified, the sockets are of family `AF_UNIX` on platforms where the family is available, and otherwise of family `AF_INET`. When `type` is not specified, it defaults to `SOCK_STREAM`.

A socket object `s` provides the following methods (out of which, those dealing with connections or requiring a connected sockets work only for `SOCK_STREAM` sockets, while the others work with both `SOCK_STREAM` and `SOCK_DGRAM` sockets). In the following table, the exact set of flags available depends on your specific platform; the `flags` values available are documented on the appropriate Unix manual page for `recv(2)` or manual page for `send(2)`:

accept

`accept()`

Blocks until a client establishes a connection to `s`, which must have been bound to an address (with a call to `s.bind`) and set to listening (with a call to `s.listen`). Returns a *new* socket object, which can be used to communicate with the other endpoint of the connection.

bind	`bind(address)` Binds `s` to a specific address. The form of the `address` argument depends on the socket's address family (see "Socket Addresses").
close	`close()` Marks the socket as closed. It does not necessarily close the connection immediately, depending on whether other references to the socket exist. If immediate closure is required, call the `s.shutdown` method first. The simplest way to ensure a socket is closed in a timely fashion is to use it in a `with` statement, since sockets are context managers.
connect	`connect(address)` Connects to a remote socket at `address`. The form of the `address` argument depends on the address family (see "Socket Addresses").
detach	`detach()` Puts the socket into closed mode, but allows the socket object to be reused for further connections.
dup	`dup()` Returns a duplicate of the socket, not inheritable by child processes.
fileno	`fileno()` Returns the socket's file descriptor.
get_inheritable	`get_inheritable()` (v3 only) Returns `True` when the socket is going to be inherited by child processes. Otherwise, returns `False`.
getpeername	`getpeername()` Returns the address of the remote endpoint to which this socket is connected.
getsockname	`getsockname()` Returns the address being used by this socket.
gettimeout	`gettimeout()` Returns the timeout associated with this socket.
listen	`listen([backlog])` Starts the socket listening for traffic on its associated endpoint. If given, the integer `backlog` argument determines how many unaccepted connections the operating system allows to queue up before starting to refuse connections.

makefile

`makefile(mode, [bufsize])` (v2)

`makefile(mode, buffering=None, *, encoding=None, newline=None)` (v3)

Returns a file object allowing the socket to be used with file-like operations such as `read` and `write`. The mode can be `'r'` or `'w'`, to which `'b'` can be added for binary transmissions. The socket must be in blocking mode; if a timeout value is set, unexpected results may be observed if a timeout occurs. Libraries intending to support both v2 and v3 are advised to omit the remaining arguments, which are not well documented and differ between versions.

recv

`recv(bufsiz, [flags])`

Receive a maximum of `bufsiz` bytes of data on the socket. Returns the received data.

recvfrom

`recvfrom(bufsiz, [flags])`

Receive a maximum of `bufsiz` bytes of data from `s`. Returns a pair `(bytes, address)` where `bytes` is the received data, and `address` the address of the counter-party socket that sent the data.

recvfrom_into

`recvfrom_into(buffer, [nbytes, [flags]])`

Receive a maximum of `nbytes` bytes of data from `s`, writing it into the given `buffer` object. Returns a two-element tuple `(nbytes, address)` where `nbytes` is the number of bytes received and `address` is the address of the socket that sent the data.

recv_into

`recv_into(buffer, [nbytes, [flags]])`

Receive a maximum of `nbytes` bytes of data from `s`, writing it into the given `buffer` object. Returns the number of bytes received.

recvmsg

`recvmsg(bufsiz, [ancbufsiz, [flags]])`

Receive a maximum of `bufsiz` bytes of data on the socket and a maximum of `ancbufsiz` of ancillary (“out-of-band”) data. Returns a four-item tuple `(data, ancdata, msg_flags, address)`, where `bytes` is the received data, `ancdata` is a list of three-item `(cmsg_level, cmsg_type, cmsg_data)` tuples representing the received ancillary data, `msg_flags` holds any flags received with the message, and `address` is the address of the counter-party socket that sent the data (if the socket is connected, this value is undefined, but the sender can be determined from the socket).

send

`send(bytes, [flags]])`

Send the given data `bytes` over the socket, which must already be connected to a remote endpoint. Returns the number of bytes sent, which should be verified: the call may not transmit all data, in which case transmission of the remainder will have to be separately requested.

sendall

`sendall(bytes, [flags])`

Send all the given data `bytes` over the socket, which must already be connected to a remote endpoint. The socket’s timeout value applies to the transmission of all the data, even if multiple transmissions are needed.

sendto

`sendto(bytes, address)` or

`sendto(bytes, flags, address)`

Transmit the `bytes` (`s` must not be connected) to the given socket address.

sendmsg

`sendmsg(buffers, [ancdata, [flags, [address]]])`

Send normal and ancillary (out-of-band) data to the connected endpoint. `buffers` should be an iterable of bytes-like objects. The `ancdata` argument should be an iterable of `(data, ancdata, msg_flags, address)` tuples representing the ancillary data, and `msg_flags` are flags values documented on the Unix manual page for the `send(2)` system call. `address` should only be provided for an unconnected socket, and determines the endpoint to which the data is sent.

sendfile

`sendfile(file, offset=0, count=None)`

Send the contents of file object `file` (which must be open in binary mode) to the connected endpoint. On platforms where `os.sendfile` is available, it's used; otherwise, the `send` call is used. If provided, `offset` determines the starting byte position in the file from which transmission begins, and `count` sets the maximum number of bytes to be transmitted. Returns the total number of bytes transmitted.

set_inheritable

`set_inheritable(flag)` (v3 only)

Determines whether the socket gets inherited by child processes, according to the Boolean value of `flag`.

setblocking

`setblocking(flag)`

Determines whether `s` operates in blocking mode (see "Socket Objects") according to the Boolean value of `flag`. `s.setblocking(True)` is equivalent to `s.settimeout(None)` and `s.set_blocking(False)` is equivalent to `s.settimeout(0.0)`.

settimeout

`settimeout(timeout)`

Establishes the mode of `s` (see "Socket Objects") according to the value of `timeout`.

shutdown

`shutdown(how)`

Shuts down one or both halves of a socket connection according to the value of the `how` argument, as detailed here:

`socket.SHUT_RD`	No further receive operations can be performed on `s`.
`socket.SHUT_WR`	No further send operations can be performed on `s`.
`socket.SHUT_RDWR`	No further receive or send operations can be performed on `s`.

A socket object `s` also has the following attributes:

`family`	An attribute that is `s`’s socket family
`type`	An attribute that is `s`’s socket type