Chapter 18. Networking Basics

Connection-oriented protocols work like making a telephone call. You request a connection to a particular *network endpoint* (equivalent to dialing somebody's phone number), and your party either answers or doesn't. If they do, you can talk to them and hear them talking back (simultaneously, if necessary), and you know that nothing is getting lost. At the end of the conversation you both say goodbye and hang up, so it's obvious something has gone wrong if that closing event doesn't occur (for example, if you just suddenly stop hearing the other party). The Transmission Control Protocol (TCP) is the main connection-oriented transport protocol of the internet, used by web browsers, secure shells, email, and many other applications.

Connectionless or *datagram* protocols work more like communicating by sending postcards. Mostly, the messages get through, but if anything goes wrong you have to be prepared to cope with the consequences—the protocol doesn't notify you whether your messages have been received, and messages can arrive out of order. For exchanging short messages and getting answers, datagram protocols have less overhead than connection-oriented ones, as long as the overall service can cope with occasional disruptions. For example, a Domain Name Service (DNS) server may fail to respond: most DNS communication was until recently connectionless. The User Datagram Protocol (UDP) is the main connectionless transport protocol for internet communications.

Nowadays, security is increasingly important: understanding the underlying basis of secure communications helps you ensure that your communications are as secure as they need to be. If this summary dissuades you from trying to implement such technology yourself without a thorough understanding of the issues and risks, it will have served a worthwhile purpose.

All communications across network interfaces exchange strings of bytes. To communicate text, or indeed most other information, the sender must encode it as bytes, which the receiver must decode. We limit our discussion in this chapter to the case of a single sender and a single receiver.

The Berkeley Socket Interface

Most networking nowadays uses *sockets.* Sockets give access to pipelines between independent endpoints, using a *transport layer protocol* to move information between those endpoints. The socket concept is general enough that the endpoints can be on the same computer, or on different computers networked together, either locally or via a wide area network.

The most frequently used transport layers today are UDP (for connectionless networking) and TCP (for connection-oriented networking); each is carried over a common Internet Protocol (IP) network layer. This stack of protocols, along with the many application protocols that run over them, is collectively known as *TCP/IP.* A good introduction is Gordon McMillan's (dated but still perfectly valid) ***Socket Programming HOWTO***.

The two most common socket families are *internet sockets* based on TCP/IP communications (available in two flavors, to accommodate the modern IPv6 and the more traditional IPv4) and *Unix sockets*, though other families are also available. Internet sockets allow communication between any two computers that can exchange IP datagrams; Unix sockets can only communicate between processes on the same Unix machine.

To support many concurrent internet sockets, the TCP/IP protocol stack uses endpoints identified by an IP address, a *port number*, and a protocol. The port numbers allow protocol handling software to distinguish between different endpoints at the same IP address using the same protocol. A connected socket is also associated with a *remote endpoint*, the counterparty socket to which it is connected and with which it can communicate.

Most Unix sockets have names in the Unix filesystem. On Linux platforms, sockets whose names begin with a zero byte live in a name pool maintained by the kernel. These are useful for communicating with a **ch-**

root-jail process, for example, where no filesystem is shared between two processes.

Both internet and Unix sockets support connectionless and connection-oriented networking, so if you write your programs carefully, they can work over either socket family. It is beyond the scope of this book to discuss other socket families, though we should mention that *raw sockets*, a subtype of the internet socket family, let you send and receive link layer packets (for example, Ethernet packets) directly. This is useful for some experimental applications and for **packet sniffing**.

After creating an internet socket, you can associate (*bind*) a specific port number with the socket (as long as that port number is not in use by some other socket). This is the strategy many servers use, offering service on so-called ***well-known port numbers*** defined by internet standards as being in the range 1–1,023. On Unix systems, *root* privileges are required to gain access to these ports. A typical client is unconcerned with the port number it uses, and so it typically requests an *ephemeral port*, assigned by the protocol driver and guaranteed to be unique on that host. There is no need to bind client ports.

Consider two processes on the same computer, each acting as a client to the same remote server. The full association for their sockets has five components, `(local_IP_address, local_port_number, protocol, remote_IP_address, remote_port_number)`. When packets arrive at the remote server, the destination, source IP address, destination port number, and protocol are the same for both clients. The guarantee of uniqueness for ephemeral port numbers lets the server distinguish between traffic from the two clients. This is how TCP/IP handles multiple conversations between the same two IP addresses.[1]

Socket Addresses

The different types of sockets use different address formats:

- Unix socket addresses are strings naming a node in the filesystem (on Linux platforms, bytestrings starting with `b'\0'` and corre-

sponding to names in a kernel table).

- IPv4 socket addresses are (*`address`*, *`port`*) pairs. The first item is an IPv4 address, the second a port number in the range 1–65,535.
- IPv6 socket addresses are four-item (*`address`*, *`port`*, *`flowinfo`*, *`scopeid`*) tuples. When providing an address as an argument, the *`flowinfo`* and *`scopeid`* items can generally be omitted, as long as the **address scope** is unimportant.

Client/Server Computing

The pattern we discuss hereafter is usually referred to as *client/server* networking, where a *server* listens for traffic on a specific endpoint from *clients* requiring the service. We do not cover *peer-to-peer* networking, which, lacking any central server, has to include the ability for peers to discover each other.

Most, though by no means all, network communication is performed using client/server techniques. The server listens for incoming traffic at a predetermined or advertised network endpoint. In the absence of such input, it does nothing, simply sitting there waiting for input from clients. Communication is somewhat different between connectionless and connection-oriented endpoints.

In connectionless networking, such as via UDP, requests arrive at a server randomly and are dealt with immediately: a response is dispatched to the requester without delay. Each request is handled on its own, usually without reference to any communications that may previously have occurred between the two parties. Connectionless networking is well suited to short-term, stateless interactions such as those required by DNS or network booting.

In connection-oriented networking, the client engages in an initial exchange with the server that effectively establishes a connection across a network pipeline between two processes (sometimes referred to as a ***virtual circuit***), across which the processes can communicate until both indicate their willingness to end the connection. In this case, serving needs to use parallelism (via a concurrency mechanism such as threads, pro-

cesses, or asynchronicity: see **Chapter 15**) to handle each incoming connection asynchronously or simultaneously. Without parallelism, the server would be unable to handle new incoming connections before earlier ones have terminated, since calls to socket methods normally *block* (meaning they pause the thread calling them until they terminate or time out). Connections are the best way to handle lengthy interactions such as mail exchanges, command-line shell interactions, or the transmission of web content, and offer automatic error detection and correction when they use TCP.

Connectionless client and server structures

The broad logic flow of a connectionless server proceeds as follows:

1. Create a socket of type `socket.SOCK_DGRAM` by calling `socket.socket`.
2. Associate the socket with the service endpoint by calling the socket's `bind` method.
3. Repeat the following steps *ad infinitum*:
    a. Request an incoming datagram from a client by calling the socket's `recvfrom` method; this call blocks until a datagram is received.
    b. Compute or look up the result.
    c. Send the result back to the client by calling the socket's `sendto` method.

The server spends most of its time in step 3a, awaiting input from clients.

A connectionless client's interaction with the server proceeds as follows:

1. Create a socket of type `socket.SOCK_DGRAM` by calling `socket.socket`.
2. Optionally, associate the socket with a specific endpoint by calling the socket's `bind` method.
3. Send a request to the server's endpoint by calling the socket's `sendto` method.

4. Await the server reply by calling the socket's `recvfrom` method; this call blocks until the response is received. It's necessary to apply a *timeout* to this call, to handle the case where a datagram goes missing and the program must either retry or abort the attempt: connectionless sockets don't guarantee delivery.
5. Use the result in the remainder of the client program's logic.

A single client program can perform several interactions with the same or multiple servers, depending on the services it needs to use. Many such interactions are hidden from the application programmer inside library code. A typical example is the resolution of a hostname to the appropriate network address, which commonly uses the `gethostbyname` library function (implemented in Python's `socket` module, discussed shortly). Connectionless interactions normally involve sending a single packet to the server and receiving a single packet in response. The main exceptions involve *streaming* protocols such as the Real-time Transport Protocol (RTP),[2] which are typically layered on top of UDP to minimize latency and delays: in streaming, many datagrams are sent and received.

Connection-oriented client and server structures

The broad flow of logic of a connection-oriented server is as follows:

1. Create a socket of type `socket.SOCK_STREAM` by calling `socket.socket`.
2. Associate the socket with the appropriate server endpoint by calling the socket's `bind` method.
3. Start the endpoint listening for connection requests by calling the socket's `listen` method.
4. Repeat the following steps *ad infinitum*:
    a. Await an incoming client connection by calling the socket's `accept` method; the server process blocks until an incoming connection request is received. When such a request arrives, a new socket object is created whose other endpoint is the client program.
    b. Create a new control thread or process to handle this specific connection, passing it the newly created socket; the main

thread of control then continues by looping back to step 4a.

c. In the new control thread, interact with the client using the new socket's `recv` and `send` methods, respectively, to read data from the client and send data to it. The `recv` method blocks until data is available from the client (or the client indicates it wishes to close the connection, in which case `recv` returns an empty result). The `send` method only blocks when the network software has so much data buffered that communication has to pause until the transport layer has emptied some of its buffer memory. When the server wishes to close the connection, it can do so by calling the socket's `close` method, optionally calling its `shutdown` method first.

The server spends most of its time in step 4a, awaiting connection requests from clients.

A connection-oriented client's overall logic is as follows:

1. Create a socket of type `socket.SOCK_STREAM` by calling `socket.socket`.
2. Optionally, associate the socket with a specific endpoint by calling the socket's `bind` method.
3. Establish a connection to the server by calling the socket's `connect` method.
4. Interact with the server using the socket's `recv` and `send` methods, respectively, to read data from the server and send data to it. The `recv` method blocks until data is available from the server (or the server indicates it wishes to close the connection, in which case the `recv` call returns an empty result). The `send` method only blocks when the network software has so much data buffered that communications have to pause until the transport layer has emptied some of its buffer memory. When the client wishes to close the connection, it can do so by calling the socket's `close` method, optionally calling its `shutdown` method first.

Connection-oriented interactions tend to be more complex than connectionless ones. Specifically, determining when to read and write data is

more complicated, because inputs must be parsed to determine when a transmission from the other end of the socket is complete. The higher-layer protocols used in connection-oriented networking accommodate this determination; sometimes this is done by indicating the data length as a part of the content, sometimes by more sophisticated methods.

The socket Module

Python's `socket` module handles networking with the socket interface. There are minor differences between platforms, but the module hides most of them, making it relatively easy to write portable networking applications.

The module defines three exception classes, all subclasses of the built-in exception class `OSError` (see **Table 18-1**).

Table 18-1. `socket` module exception classes

`herror`	Identifies hostname resolution errors: e.g., `socket.gethostbyname` cannot convert a name to a network address, or `socket.gethostbyaddr` can find no hostname for a network address. The accompanying value is a two-element tuple (*`h_errno, string`*), where *`h_errno`* is the integer error number from the operating system, and *`string`* is a description of the error.
`gaierror`	Identifies addressing errors encountered in `socket.getaddrinfo` or `socket.getnameinfo`.
`timeout`	Raised when an operation takes longer than the timeout limit (as per `socket.setdefaulttimeout`, overridable on a per-socket basis).

The module defines many constants. The most important of these are the address families (`AF_*`) and the socket types (`SOCK_*`) listed in **Table 18-2**, members of `IntEnum` collections. The module also defines many other constants used to set socket options, but the documentation does not de-

fine them fully: to use them you must be familiar with documentation for the C sockets library and system calls.

Table 18-2. Important constants defined in the `socket` module

`AF_BLUETOOTH`	Used to create sockets of the Bluetooth address family, used in mobile and Personal Area Network (PAN) applications.
`AF_CAN`	Used to create sockets for the Controller Area Network (CAN) address family, widely used in automation, automotive, and embedded device applications.
`AF_INET`	Used to create sockets of the IPv4 address family.
`AF_INET6`	Used to create sockets of the IPv6 address family.
`AF_UNIX`	Used to create sockets of the Unix address family. This constant is only defined on platforms that make Unix sockets available.
`SOCK_DGRAM`	Used to create connectionless sockets, which provide best-effort message delivery without connection capabilities or error detection.
`SOCK_RAW`	Used to create sockets that give direct access to the link layer drivers; typically used to implement lower-level network features.
`SOCK_RDM`	Used to create reliable connectionless message sockets used in the Transparent Inter Process Communication (TIPC) protocol.
`SOCK_SEQPACKET`	Used to create reliable connection-oriented message sockets used in the TIPC protocol.

`SOCK_STREAM`	Used to create connection-oriented sockets, which provide full error detection and correction facilities.

The module defines many functions to create sockets, manipulate address information, and assist with standard representations of data. We do not cover all of them in this book, as the socket module's **documentation** is fairly comprehensive; we deal only with those that are essential in writing networked applications.

The `socket` module contains many functions, most of which are only used in specific situations. For example, when communication takes place between network endpoints, the computers at either end might have architectural differences and represent the same data in different ways, so there are functions to handle translation of a limited number of data types to and from a network-neutral form. **Table 18-3** lists a few of the more generally applicable functions this module provides.

Table 18-3. Useful functions of the `socket` module

`getaddrinfo`	`socket.getaddrinfo(`*`host`*`, `*`port`*`, family=0, type=0, proto=0, flags=0)` Takes a *`host`* and *`port`* and returns a list of five-item tuples of the form `(family, type, proto,` *`canonical_name, socket`*`)` usable to create a socket connection to a specific service. *`canonical_name`* is an empty string unless the `socket.AI_CANONNAME` bit is set in the `flags` argument. When you pass a hostname rather than an IP address, `getaddrinfo` returns a list of tuples, one per IP address associated with the name.
`getdefaulttimeout`	`socket.getdefaulttimeout()` Returns the default timeout value in seconds for socket operations, or **`None`** if no value has been set. Some functions let you specify explicit timeouts.

`getfqdn`	`socket.getfqdn([`*`host`*`])` Returns the fully qualified domain name associated with a hostname or network address (by default, that of the computer on which you call it).
`gethostbyaddr`	`socket.gethostbyaddr(`*`ip_address`*`)` Takes a string containing an IPv4 or IPv6 address and returns a three-item tuple of the form (*`hostname`*`, `*`aliaslist`*`, `*`ipaddrlist`*`)`. *`hostname`* is the canonical name for the IP address, *`aliaslist`* is a list of alternative names, and *`ipaddrlist`* is a list of IPv4 and IPv6 addresses.
`gethostbyname`	`socket.gethostbyname(hostname)` Returns a string containing the IPv4 address associated with the given hostname. If called with an IP address, returns that address. This function does not support IPv6: use `getaddrinfo` for IPv6.
`getnameinfo`	`socket.getnameinfo(`*`sock_addr`*`, flags=0)` Takes a socket address and returns a (*`host`*`, `*`port`*`)` pair. Without `flags`, *`host`* is an IP address and *`port`* is an `int`.
`setdefault timeout`	`socket.setdefaulttimeout(`*`timeout`*`)` Sets sockets' default timeout as a value in floating-point seconds. Newly created sockets operate in the mode determined by the *`timeout`* value, as discussed in the next section. Pass *`timeout`* as **None** to cancel the implicit use of timeouts on subsequently created sockets.

Socket Objects

The socket object is the primary means of network communication in Python. A new socket is also created when a `SOCK_STREAM` socket accepts a connection, each such socket being used to communicate with the relevant client.

SOCKET OBJECTS AND WITH STATEMENTS

Every socket object is a context manager: you can use any socket object in a **`with`** statement to ensure proper termination of the socket at exit from the statement's body. For further details, see **"The with Statement and Context Managers"**.

There are several ways to create a socket, as detailed in the next section. Sockets can operate in three different modes, shown in **Table 18-4**, according to the timeout value, which can be set in different ways:

- By providing the timeout value as an argument on socket creation
- By calling the socket object's `settimeout` method
- According to the `socket` module's default timeout value as returned by the `socket.getdefaulttimeout` function

The timeout values to establish each possible mode are listed in **Table 18-4**.

Table 18-4. Timeout values and their associated modes

None	Sets *blocking* mode. Each operation suspends the thread (*blocks*) until the operation completes, unless the operating system raises an exception.
`0`	Sets *nonblocking* mode. Each operation raises an exception when it cannot be completed immediately, or when an error occurs. Use the **`selectors` module** to find out whether an operation can be completed immediately.

`>0.0`	Sets *timeout* mode. Each operation blocks until complete, or the timeout elapses (in which case it raises a `socket.timeout` exception), or an error occurs.

Socket objects represent network endpoints. The `socket` module supplies several functions to create a socket (see **Table 18-5**).

Table 18-5. Socket creation functions

`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` (in unspecified order), then a connection is attempted to each returned address in turn—a convenient way to create client programs able to use either IPv6 or IPv4. The *`timeout`* argument, if given, specifies the connection timeout in seconds and thereby sets the socket's mode (see **Table 18-4**); when not present, the `socket.getdefaulttimeout` function is called to determine the value. The *`source_address`* argument, if given, must also be a `(`*`host, port`*`)` pair 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). Child processes do not inherit the socket thus created. 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.
`socketpair`	`socketpair([`*`family`*`[, `*`type`*`[, `*`proto`*`]]])` Returns a connected pair of sockets of the given address family, socket type, and (for CAN sockets only) protocol. When *`family`* is not specified, the sockets are of family `AF_UNIX` on platforms where the family is available; otherwise, they are of family `AF_INET`. When *`type`* is not specified, it defaults to `SOCK_STREAM`.

A socket object *`s`* provides the methods listed in **Table 18-6**. Those dealing with connections or requiring connected sockets work only for `SOCK_STREAM` sockets, while the others work with both `SOCK_STREAM` and `SOCK_DGRAM` sockets. For methods that take a *`flags`* argument, the exact set of flags available depends on your specific platform (the values available are documented on the Unix manual pages for **`recv(2)`** and **`send(2)`** and in the **Windows docs**); if omitted, *`flags`* defaults to `0`.

Table 18-6. Methods of an instance `s` of `socket`

`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. Calling `s.close` 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 (by calling `connect` again).
`dup`	`dup()` Returns a duplicate of the socket, not inheritable by child processes.
`fileno`	`fileno()` Returns the socket's file descriptor.
`getblocking`	`getblocking()` Returns **`True`** if the socket is set to be blocking, either with a call to `s.setblocking(`**`True`**`)` or `s.settimeout(`**`None`**`)`. Otherwise, returns **`False`**.

`get_inheritable`	`get_inheritable()` Returns **`True`** when the socket is able 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`*`, buffering=`**`None`**`, *, encoding=`**`None`**`, newline=`**`None`**`)` Returns a file object allowing the socket to be used with file-like operations such as `read` and `write`. The arguments are like those for the built-in `open` function (see **"Creating a File Object with open"**). *`mode`* can be `'r'` or `'w'`; `'b'` can be added for binary transmission. The socket must be in blocking mode; if a timeout value is set, unexpected results may be observed if a timeout occurs.

recv	recv(*bufsiz*[, *flags*]) Receives and returns a maximum of *bufsiz* bytes of data from the socket *s*.
recvfrom	recvfrom(*bufsiz*[, *flags*]) Receives a maximum of *bufsiz* bytes of data from *s*. Returns a pair (*bytes*, *address*): *bytes* is the received data, *address* the address of the counterparty socket that sent the data.
recvfrom_into	recvfrom_into(*buffer*[, *nbytes*[, *flags*]]) Receives a maximum of *nbytes* bytes of data from *s*, writing it into the given *buffer* object. If *nbytes* is omitted or 0, len(*buffer*) is used. Returns a pair (*nbytes*, *address*): *nbytes* is the number of bytes received, *address* the address of the counterparty socket that sent the data (*_*into* functions can be faster than "plain" ones allocating new buffers).
recv_into	recv_into(*buffer*[, *nbytes*[, *flags*]]) Receives a maximum of *nbytes* bytes of data from *s*, writing it into the given *buffer* object. If *nbytes* is omitted or 0, len(*buffer*) is used. Returns the number of bytes received.
recvmsg	recvmsg(*bufsiz*[, *ancbufsiz*[, *flags*]]) Receives a maximum of *bufsiz* bytes of data on the socket and a maximum of *ancbufsiz* bytes 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 (documented on the Unix manual page for

	the **recv(2)** system call or in the **Windows docs**), and *address* is the address of the counterparty 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*]) Sends the given data *bytes* over the socket, which must already be connected to a remote endpoint. Returns the number of bytes sent, which you should check: the call may not transmit all data, in which case transmission of the remainder will have to be separately requested.
sendall	sendall(*bytes*[, *flags*]) Sends 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.
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. offset, if any, determines the starting byte position in the file from which transmission begins; count sets the maximum number of bytes to transmit. Returns the total number of bytes transmitted.
sendmsg	sendmsg(*buffers*[, *ancdata*[, *flags*[, *address*]]]) Sends 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. *msg_flags* are flags documented on the Unix manual page for the `send(2)` system call or in the **Windows docs**. *address* should only be provided for an unconnected socket, and determines the endpoint to which the data is sent.
`sendto`	`sendto(`*bytes*`,[`*flags*`,]`*address*`)` Transmits the *bytes* (*s* must not be connected) to the given socket address, and returns the number of bytes sent. The optional *flags* argument has the same meaning as for `recv`.
`setblocking`	`setblocking(`*flag*`)` Determines whether *s* operates in blocking mode (see **"Socket Objects"**), according to the truth value of *flag*. `s.setblocking(True)` works like `s.settimeout(None)`; `s.set_blocking(False)` works like `s.settimeout(0.0)`.
`set_inheritable`	`set_inheritable(`*flag*`)` Determines whether the socket gets inherited by child processes, according to the truth value of *flag*.
`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_RDWR`
: No further receive or send operations can be performed on *s*.

`socket.SHUT_WR`
: No further send operations can be performed on *s*.

A socket object *s* also has the attributes `family` (*s*'s socket family) and `type` (*s*'s socket type).

A Connectionless Socket Client

Consider a simplistic packet-echo service, where a client sends text encoded in UTF-8 to a server, which sends the same information back to the client. In a connectionless service, all the client has to do is send each chunk of data to the defined server endpoint:

```
import socket

UDP_IP = 'localhost'
UDP_PORT = 8883
MESSAGE = """\
This is a bunch of lines, each
of which will be sent in a single
UDP datagram. No error detection
or correction will occur.
Crazy bananas! £€ should go through."""

server = UDP_IP, UDP_PORT
encoding = 'utf-8'
with socket.socket(socket.AF_INET,    # IPv4
                   socket.SOCK_DGRAM, # UDP
                  ) as sock:
    for line in MESSAGE.splitlines():
```

```
        data = line.encode(encoding)
        bytes_sent = sock.sendto(data, server)
        print(f'SENT {data!r} ({bytes_sent} of {len(data)})'
                     f' to {server}')
        response, address = sock.recvfrom(1024)  # buffer size: 1024
        print(f'RCVD {response.decode(encoding)!r}'
              f' from {address}')

print('Disconnected from server')
```

Note that the server only performs a bytes-oriented echo function. The client, therefore, encodes its Unicode data into bytestrings, and decodes the bytestring responses received from the server back into Unicode text using the same encoding.

A Connectionless Socket Server

A server for the packet-echo service described in the previous section is also quite simple. It binds to its endpoint, receives packets (datagrams) at that endpoint, and returns to the client sending each datagram a packet with exactly the same data. The server treats all clients equally and does not need to use any kind of concurrency (though this last handy characteristic might not hold for a service where request handling takes more time).

The following server works, but offers no way to terminate the service other than by interrupting it (typically from the keyboard, with Ctrl-C or Ctrl-Break):

```
import socket

UDP_IP = 'localhost'
UDP_PORT = 8883
with socket.socket(socket.AF_INET,    # IPv4
                   socket.SOCK_DGRAM  # UDP
                   ) as sock:
    sock.bind((UDP_IP, UDP_PORT))
    print(f'Serving at {UDP_IP}:{UDP_PORT}')
```

```
    while True:
        data, sender_addr = sock.recvfrom(1024)  # 1024-byte buffer
        print(f'RCVD {data!r}) from {sender_addr}')
        bytes_sent = sock.sendto(data, sender_addr)
        print(f'SENT {data!r} ({bytes_sent}/{len(data)})'
              f' to {sender_addr}')
```

Neither is there any mechanism to handle dropped packets and similar network problems; this is often acceptable in simple services.

You can run the same programs using IPv6: simply replace the socket type `AF_INET` with `AF_INET6`.

A Connection-Oriented Socket Client

Now consider a simplistic connection-oriented "echo-like" protocol: a server lets clients connect to its listening socket, receives arbitrary bytes from them, and sends back to each client the same bytes that client sent to the server, until the client closes the connection. Here's an example of an elementary test client:[3]

```
import socket

IP_ADDR = 'localhost'
IP_PORT = 8881
MESSAGE = """\
A few lines of text
including non-ASCII characters: €£
to test the operation
of both server
and client."""

encoding = 'utf-8'
with socket.socket(socket.AF_INET,     # IPv4
                   socket.SOCK_STREAM  # TCP
                   ) as sock:
    sock.connect((IP_ADDR, IP_PORT))
    print(f'Connected to server {IP_ADDR}:{IP_PORT}')
    for line in MESSAGE.splitlines():
```

```
            data = line.encode(encoding)
            sock.sendall(data)
            print(f'SENT {data!r} ({len(data)})')
            response, address = sock.recvfrom(1024)  # buffer size: 1024
            print(f'RCVD {response.decode(encoding)!r}'
                  f' ({len(response)}) from {address}')

    print('Disconnected from server')
```

Note that the data is text, so it must be encoded with a suitable representation. We chose the usual suspect, UTF-8. The server works in terms of bytes (since it is bytes, aka octets, that travel on the network); the received bytes object gets decoded with UTF-8 back into Unicode text before printing. Any other suitable codec could be used instead: the key point is that text must be encoded before transmission and decoded after reception. The server, working in terms of bytes, does not even need to know which encoding is being used, except maybe for logging purposes.

A Connection-Oriented Socket Server

Here is a simplistic server corresponding to the testing client shown in the previous section, using multithreading via `concurrent.futures` (covered in **"The concurrent.futures Module"**):

```
import concurrent
import socket

IP_ADDR = 'localhost'
IP_PORT = 8881

def handle(new_sock, address):
    print('Connected from', address)
    with new_sock:
        while True:
            received = new_sock.recv(1024)
            if not received:
                break
            s = received.decode('utf-8', errors='replace')
            print(f'Recv: {s!r}')
```

```
                new_sock.sendall(received)
                print(f'Echo: {s!r}')
        print(f'Disconnected from {address}')

with socket.socket(socket.AF_INET,     # IPv4
                   socket.SOCK_STREAM  # TCP
                   ) as servsock:
    servsock.bind((IP_ADDR, IP_PORT))
    servsock.listen(5)
    print(f'Serving at {servsock.getsockname()}')
    with cconcurrent.futures.ThreadPoolExecutor(20) as e:
        while True:
            new_sock, address = servsock.accept()
            e.submit(handle, new_sock, address)
```

This server has its limits. In particular, it runs only 20 threads, so it cannot simultaneously serve more than 20 clients; any further client trying to connect while 20 others are already being served waits in `servsock`'s listening queue. Should that queue fill up with five clients waiting to be accepted, further clients attempting connection get rejected outright. This server is intended just as an elementary example for demonstration purposes, not as a solid, scalable, or secure system.

As before, the same programs can be run using IPv6 by replacing the socket type `AF_INET` with `AF_INET6`.

Transport Layer Security

Transport Layer Security (TLS), the successor of Secure Sockets Layer (SSL), provides privacy and data integrity over TCP/IP, helping you defend against server impersonation, eavesdropping on the bytes being exchanged, and malicious alteration of those bytes. For an introduction to TLS, we recommend the extensive **Wikipedia entry**.

In Python, you can use TLS via the `ssl` module of the standard library. To use `ssl` well, you need a good grasp of its rich **online docs**, as well as a deep and broad understanding of TLS itself (the Wikipedia article, excellent and vast as it is, can only begin to cover this large, difficult subject).

In particular, you must study and thoroughly understand the **security considerations section of the online docs**, as well as all the materials found at the many links helpfully offered in that section.

If these warnings make it sound as though a perfect implementation of security precautions is a daunting task, that's because it *is.* In security, you're pitting your wits and skills against those of sophisticated attackers who may be more familiar with the nooks and crannies of the problems involved: they specialize in finding workarounds and breaking in, while (usually) your focus is not exclusively on such issues—rather, you're trying to provide some useful services in your code. It's risky to see security as an afterthought or a secondary point—it *has* to be front and center throughout, to win said battle of skills and wits.

That said, we strongly recommend that all readers undertake the study of TLS mentioned above—the better all developers understand security considerations, the better off we all are (except, we guess, the security-breaker wannabes!).

Unless you have acquired a really deep and broad understanding of TLS and Python's `ssl` module (in which case, you'll know what exactly to do—better than we possibly could!), we recommend using an `SSLContext` instance to hold all the details of your use of TLS. Build that instance with the `ssl.create_default_context` function, add your certificate if needed (it *is* needed if you're writing a secure server), then use the instance's `wrap_socket` method to wrap (almost[4]) every `socket.socket` instance you make into an instance of `ssl.SSLSocket`—behaving almost identically to the socket object it wraps, but nearly transparently adding security checks and validation "on the side."

The default TLS contexts strike a good compromise between security and broad usability, and we recommend you stick with them (unless you're knowledgeable enough to fine-tune and tighten security for special needs). If you need to support outdated counterparts that are unable to use the most recent, most secure implementations of TLS, you may feel tempted to learn just enough to relax your security demands. Do that at

your own risk—we most definitely *don't* recommend wandering into such territory!

In the following sections, we cover the minimal subset of `ssl` you need to be familiar with if you just want to follow our recommendations. But even if that is the case, *please* also read up on TLS and `ssl`, just to gain some background knowledge about the intricate issues involved. It may stand you in good stead one day!

SSLContext

The `ssl` module supplies an `ssl.SSLContext` class, whose instances hold information about TLS configuration (including certificates and private keys) and offer many methods to set, change, check, and use that information. If you know exactly what you're doing, you can manually instantiate, set up, and use your own `SSLContext` instances for your own specialized purposes.

However, we recommend instead that you instantiate an `SSLContext` using the well-tuned function `ssl.create_default_context`, with a single argument: `ssl.Purpose.CLIENT_AUTH` if your code is a server (and thus may need to authenticate clients), or `ssl.Purpose.SERVER_AUTH` if your code is a client (and thus definitely needs to authenticate servers). If your code is both a client to some servers and a server to other clients (as, for example, some internet proxies are), then you'll need two instances of `SSLContext`, one for each purpose.

For most client-side uses, your `SSLContext` is ready. If you're coding a server, or a client for one of the rare servers that require TLS authentication of the clients, you need to have a certificate file and a key file (see the **online docs** to learn how to obtain these files). Add them to the `SSLContext` instance (so that counterparties can verify your identity) by passing the paths to the certificate and key files to the `load_cert_chain` method with code like the following:

```
ctx = ssl.create_default_context(ssl.Purpose.CLIENT_AUTH)
ctx.load_cert_chain(certfile='mycert.pem', keyfile='mykey.key')
```

Once your context instance *ctx* is ready, if you're coding a client, just call *ctx*.wrap_socket to wrap any socket you're about to connect to a server, and use the wrapped result (an instance of ssl.SSLSocket) instead of the socket you just wrapped. For example:

```
sock = socket.socket(socket.AF_INET)
sock = ctx.wrap_socket(sock, server_hostname='www.example.com')
sock.connect(('www.example.com', 443))
# use 'sock' normally from here on
```

Note that, in the client case, you should also pass wrap_socket a server_hostname argument corresponding to the server you're about to connect to; this way, the connection can verify that the identity of the server you end up connecting to is indeed correct, an absolutely crucial security step.

Server-side, *don't* wrap the socket that you are binding to an address, listening on, or accepting connections on; just wrap the new socket that accept returns. For example:

```
sock = socket.socket(socket.AF_INET)
sock.bind(('www.example.com', 443))
sock.listen(5)
while True:
    newsock, fromaddr = sock.accept()
    newsock = ctx.wrap_socket(newsock, server_side=True)
    # deal with 'newsock' as usual; shut down, then close it, when done
```

In this case, you need to pass wrap_socket the argument server_side=**True** so it knows that you're on the server side of things.

Again, we recommend consulting the online docs—particularly the **examples**—for better understanding, even if you stick to just this simple subset of `ssl` operations.

1 When you code an application program, you normally use sockets through higher-abstraction layers, such as those covered in **Chapter 19**.

2 And the relatively newfangled multiplexed connections transport protocol **QUIC**, supported in Python by third-party **`aioquic`**.

3 This client example isn't secure; see **"Transport Layer Security"** for an introduction to making it secure.

4 We say "almost" because, when you code a server, you don't wrap the socket you bind, listen on, and accept connections from.