

HOWTO

3.7.2

**Guido van Rossum
and the Python development team**

12, 2019

Python Software Foundation
Email: docs@python.org

Contents

1	1.1	2
2	2.1	2
3	3.1	4
4	Disconnecting	4
	4.1 When Sockets Die	4
5	Non-blocking Sockets	4

Gordon McMillan

1

INET	IPv4	99%	STREAM	TCP	-	HOWTO
STREAM						

1.1

IPC () IPC
BSD Unix ——— INET

2

:

```
# create an INET, STREAMing socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# now connect to the web server on port 80 - the normal http port
s.connect(("www.python.org", 80))
```

:

```
# create an INET, STREAMing socket
serversocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# bind the socket to a public host, and a well-known port
serversocket.bind((socket.gethostname(), 80))
# become a server socket
serversocket.listen(5)
```

```
socket.gethostname()                      s.bind(('localhost', 80))                      s.bind(('127.0.0.1',
80))                      s.bind('', 80))                      4
HTTP, SNMP                      4
listen                      5
80                      :
```

```
while True:
    # accept connections from outside
    (clientsocket, address) = serversocket.accept()
    # now do something with the clientsocket
    # in this case, we'll pretend this is a threaded server
    ct = client_thread(clientsocket)
    ct.run()
```

```
3                      -                      select
                                                 connect()                      -
```

2.1

IPC AF_INET 'localhost'

:

multiprocessing IPC API

3

```

        send      recv      read      write      Java      flush
        write      flush      read
    - send      recv      send      recv
recv      0
HTTP      0
        :abbr:'EOT ( )'      send      recv      0      recv
:

```

```

class MySocket:
    """demonstration class only
    - coded for clarity, not efficiency
    """

    def __init__(self, sock=None):
        if sock is None:
            self.sock = socket.socket(
                socket.AF_INET, socket.SOCK_STREAM)
        else:
            self.sock = sock

    def connect(self, host, port):
        self.sock.connect((host, port))

    def mysend(self, msg):
        totalsent = 0
        while totalsent < MSGLEN:
            sent = self.sock.send(msg[totalsent:])
            if sent == 0:
                raise RuntimeError("socket connection broken")
            totalsent = totalsent + sent

    def myreceive(self):
        chunks = []
        bytes_recd = 0
        while bytes_recd < MSGLEN:
            chunk = self.sock.recv(min(MSGLEN - bytes_recd, 2048))
            if chunk == b'':
                raise RuntimeError("socket connection broken")
            chunks.append(chunk)
            bytes_recd = bytes_recd + len(chunk)
        return b''.join(chunks)

```

```

- Python      len()      \ 0      C      \ 0      strlen
recv -      4096      8192
recv

```

```

send          5          recv          5          "recv" -

```

3.1

```

00      1      16      32      ——— Motorola      00 01      16      1      Intel      DEC      ———      01
                                ntohl, htonl, ntohs, htons      n      network h      host s      short l
long
      32      ascii      long      0      1      "0"

```

4 Disconnecting

Strictly speaking, you're supposed to use `shutdown` on a socket before you `close` it. The `shutdown` is an advisory to the socket at the other end. Depending on the argument you pass it, it can mean "I'm not going to send anymore, but I'll still listen", or "I'm not listening, good riddance!". Most socket libraries, however, are so used to programmers neglecting to use this piece of etiquette that normally a `close` is the same as `shutdown(); close()`. So in most situations, an explicit `shutdown` is not needed.

One way to use `shutdown` effectively is in an HTTP-like exchange. The client sends a request and then does a `shutdown(1)`. This tells the server "This client is done sending, but can still receive." The server can detect "EOF" by a receive of 0 bytes. It can assume it has the complete request. The server sends a reply. If the `send` completes successfully then, indeed, the client was still receiving.

Python takes the automatic shutdown a step further, and says that when a socket is garbage collected, it will automatically do a `close` if it's needed. But relying on this is a very bad habit. If your socket just disappears without doing a `close`, the socket at the other end may hang indefinitely, thinking you're just being slow. *Please close* your sockets when you're done.

4.1 When Sockets Die

Probably the worst thing about using blocking sockets is what happens when the other side comes down hard (without doing a `close`). Your socket is likely to hang. TCP is a reliable protocol, and it will wait a long, long time before giving up on a connection. If you're using threads, the entire thread is essentially dead. There's not much you can do about it. As long as you aren't doing something dumb, like holding a lock while doing a blocking read, the thread isn't really consuming much in the way of resources. Do *not* try to kill the thread - part of the reason that threads are more efficient than processes is that they avoid the overhead associated with the automatic recycling of resources. In other words, if you do manage to kill the thread, your whole process is likely to be screwed up.

5 Non-blocking Sockets

If you've understood the preceding, you already know most of what you need to know about the mechanics of using sockets. You'll still use the same calls, in much the same ways. It's just that, if you do it right, your app will be almost inside-out.

In Python, you use `socket.setblocking(0)` to make it non-blocking. In C, it's more complex, (for one thing, you'll need to choose between the BSD flavor `O_NONBLOCK` and the almost indistinguishable Posix

flavor `O_NDELAY`, which is completely different from `TCP_NODELAY`), but it's the exact same idea. You do this after creating the socket, but before using it. (Actually, if you're nuts, you can switch back and forth.)

The major mechanical difference is that `send`, `recv`, `connect` and `accept` can return without having done anything. You have (of course) a number of choices. You can check return code and error codes and generally drive yourself crazy. If you don't believe me, try it sometime. Your app will grow large, buggy and suck CPU. So let's skip the brain-dead solutions and do it right.

Use `select`.

In C, coding `select` is fairly complex. In Python, it's a piece of cake, but it's close enough to the C version that if you understand `select` in Python, you'll have little trouble with it in C:

```
ready_to_read, ready_to_write, in_error = \
    select.select(
        potential_readers,
        potential_writers,
        potential_errs,
        timeout)
```

You pass `select` three lists: the first contains all sockets that you might want to try reading; the second all the sockets you might want to try writing to, and the last (normally left empty) those that you want to check for errors. You should note that a socket can go into more than one list. The `select` call is blocking, but you can give it a timeout. This is generally a sensible thing to do - give it a nice long timeout (say a minute) unless you have good reason to do otherwise.

In return, you will get three lists. They contain the sockets that are actually readable, writable and in error. Each of these lists is a subset (possibly empty) of the corresponding list you passed in.

If a socket is in the output readable list, you can be as-close-to-certain-as-we-ever-get-in-this-business that a `recv` on that socket will return *something*. Same idea for the writable list. You'll be able to send *something*. Maybe not all you want to, but *something* is better than nothing. (Actually, any reasonably healthy socket will return as writable - it just means outbound network buffer space is available.)

If you have a "server" socket, put it in the `potential_readers` list. If it comes out in the readable list, your `accept` will (almost certainly) work. If you have created a new socket to `connect` to someone else, put it in the `potential_writers` list. If it shows up in the writable list, you have a decent chance that it has connected.

Actually, `select` can be handy even with blocking sockets. It's one way of determining whether you will block - the socket returns as readable when there's something in the buffers. However, this still doesn't help with the problem of determining whether the other end is done, or just busy with something else.

Portability alert: On Unix, `select` works both with the sockets and files. Don't try this on Windows. On Windows, `select` works with sockets only. Also note that in C, many of the more advanced socket options are done differently on Windows. In fact, on Windows I usually use threads (which work very, very well) with my sockets.