

---

# Programação de Soquetes

*Release 3.11.13*

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

julho 07, 2025

Python Software Foundation  
Email: [docs@python.org](mailto:docs@python.org)

## Sumário

<b>1</b>	<b>Soquetes</b>	<b>2</b>
1.1	História . . . . .	2
<b>2</b>	<b>Criando um Soquete</b>	<b>2</b>
2.1	IPC . . . . .	3
<b>3</b>	<b>Usando um Soquete</b>	<b>3</b>
3.1	Dados Binários . . . . .	5
<b>4</b>	<b>Desconectando</b>	<b>5</b>
4.1	Quando o Soquete Morre . . . . .	6
<b>5</b>	<b>Soquetes não-bloqueantes</b>	<b>6</b>

---

### Autor

Gordon McMillan

### Resumo

Os soquetes são usados em quase todos os lugares, mas são uma das tecnologias mais mal compreendidas. Esta é uma visão geral contendo 10.000 pontos a respeito dos soquetes. Não é realmente um tutorial - ainda terá o trabalho para tornar tudo operacionais. Não abrange os pontos finos (e há muitos deles), mas espero que lhe dê uma base suficiente para começar a utilizar de maneira correta.

# 1 Soquetes

Só trataremos dos soquetes INET (ou seja, IPv4), no entanto, os mesmos representam ao menos 99% dos soquetes que estão atualmente em uso. Só estudaremos os soquetes STREAM (ou seja, TCP) - a menos que você realmente saiba o que está fazendo (nesse caso, este documento não é para você!), terás um melhor conhecimento sobre o comportamento e o desempenho dos soquetes STREAM do que qualquer outra coisa. Tentarei aclarar o mistério sobre o que realmente é um soquete, bem como, apresentarei dicas de como trabalhar com soquetes bloqueantes e os não bloqueantes. Começaremos o estudo falando da razão pela qual bloquear um soquete. Precisas saber como os mesmos (os soquetes bloqueantes) antes de estudar os não bloqueantes

Parte do problema em compreender o funcionamento se dá pelo fato de que o “soquete” pode ter significados que são sutilmente diferentes, dependendo do contexto. Então, primeiro, vamos fazer uma distinção entre um soquete “cliente” - o ponto final de uma conversa e um soquete “servidor”, que mais parece como um operador de painel. O aplicativo Cliente (seu navegador, por exemplo) usa soquetes “cliente” exclusivos; O servidor Web com o qual conversamos faz uso de soquetes “servidor” e soquetes de “cliente”.

## 1.1 História

Das várias formas de IPC (Inter Process Communication), os soquetes são, de longe, os mais populares. Em qualquer plataforma, muito provável existirão outras formas de IPC que sejam mais rápidas, porém, para a comunicação entre plataformas, os soquetes são sobre o único jogo na cidade.

They were invented in Berkeley as part of the BSD flavor of Unix. They spread like wildfire with the internet. With good reason — the combination of sockets with INET makes talking to arbitrary machines around the world unbelievably easy (at least compared to other schemes).

## 2 Criando um Soquete

De forma geral, quando clicastes no link que te trouxe até esta página (documentação do Python), o que o teu navegador fez, foi o seguinte:

```
# 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))
```

Quando a connect (conexão) foi estabelecida, o soquete s pode ser utilizado para enviar uma solicitação de texto para a página. O mesmo soquete é que irá ler a resposta e, em seguida, o mesmo será destruído. Isso mesmo, será destruído. Os soquetes de Clientes normalmente são usados apenas numa única transação (troca) (ou um pequeno conjunto sequencial de transações).

O que acontece no lado do servidor Web é um pouco mais complexo. Primeiro, o Servidor Web cria um “soquete tipo servidor”:

```
# 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)
```

Algumas coisas que deves observar: usamos socket.gethostname() para que o soquete esteja visível ao mundo exterior. Se tivéssemos usado s.bind(('localhost', 80)) ou s.bind(('127.0.0.1', 80)) ainda teríamos um soquete do tipo “servidor”, mas o mesmo só estaria visível dentro do computador em que está sendo executado. s.bind('', 80) determina que o soquete estará acessível por qualquer computador que possuas o endereço IP do computador.

Uma segunda coisa que precisamos observar é: as portas baixas, normalmente estão reservadas para serviços “bem conhecidos”, tais como (HTTP, SNMP etc). Caso esteja brincando, utilize um número alto (números contendo 4 dígitos).

Por fim, o argumento “listen” diz à biblioteca de soquetes que queremos enfileirar no máximo 5 requisições de conexão (normalmente o máximo) antes de recusar começar a recusar conexões externas. Caso o resto do código esteja escrito corretamente, isso deverá ser o suficiente.

Agora que temos um soquete tipo “servidor”, que está ouvindo a porta 80, podemos entrar no laço de repetição principal do servidor web:

```
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()
```

Na verdade, existem 3 formas gerais nas quais este loop pode ser implementado - despachando um handle que lide com o `clientsocket`, criando um novo processo que funcione com o `clientsocket`, ou reestruture o aplicativo para trabalhar com soquetes não bloqueáveis e multiplexar entre os nossos soquetes tipo “servidor” e qualquer cliente ativo que faça uso do `select`. Trataremos mais sobre isso posteriormente. O importante que temos que saber neste momento é isso: isso é *tudo* o que um soquete tipo “Servidor” irá fazer. O mesmo não enviar quaisquer dados. Não recebe dados. Apenas produz soquetes “cliente”. Cada `clientsocket` será criado em resposta a algum *outro* soquete “cliente” que fará uma `connect()` ao um determinado host numa determinada porta que estamos vinculados. Será dessa forma que criamos aquele “cliente”, voltaremos a ouvir mais conexões. Os dois “clientes” são livres para conversar - eles estão usando uma porta alocada dinamicamente que será descartada quando a conversa terminar.

## 2.1 IPC

Se precisamos que o IPC seja rápido entre dois processos numa máquina, deverás procurar por pipes ou compartilhamento de memória. Se decidires usar os soquetes `AF_INET`, vincule o soquete “servidor” a `'localhost'`. Na maioria das plataformas, isso levará um atalho em torno de algumas camadas de código de rede e funcionará um pouco mais rápido.

### Ver também:

A módulo `multiprocessing` faz a integração do IPC de forma multiplataforma numa API de nível mais alto.

## 3 Usando um Soquete

A primeira coisa que precisamos observar, é que o soquete “cliente” do navegador e o soquete “cliente” do servidor Web são “animais” idênticos. Ou seja, esta é uma conversa “peer to peer”. Ou, em outras palavras, *como designer, terás que decidir quais são as regras da etiqueta para uma conversa*. Normalmente, o soquete `connect`ando inicia a conversa, enviando uma solicitação ou talvez um sinal. Mas essa é uma decisão de design - não é uma regra de soquetes.

Agora, há dois conjuntos de verbos que podemos usar na comunicação. Podemos usar `send` e `recv`, ou podemos transformar o seu soquete cliente numa “besta” semelhante a um arquivo e usar `read` e `write`. O último é o modo como Java apresenta seus soquetes. Não tratarei disso aqui, exceto para avisar que precisamos usar o `flush` com soquetes. Estes são “arquivos” em buffer e um erro comum é “escrever” algo e, em seguida, “ler” uma resposta. Sem que haja um `flush` lá dentro, poderás esperar eternamente pela resposta, porque o pedido ainda poderá estar no buffer de saída.

Agora, chegamos ao principal obstáculo no uso dos soquetes - `send` e `recv` operam nos buffers de rede. Eles não lidam necessariamente com todos os bytes que você os entrega (ou esperam deles), porque o foco principal deles é lidar com os buffers de rede. Em geral, os mesmos retornam quando os buffers de rede associados foram preenchidos

(`send`) ou esvaziados (`recv`). Eles então lhe dizem quantos bytes eles trataram. É *sua* responsabilidade chamá-los novamente até que a sua mensagem tenha sido completamente tratada.

Quando um `recv` retornar 0 bytes, significa que o outro lado finalizou a conexão (ou está no processo de fechamento) da conexão. Não receberás mais dados nesta conexão. Pra sempre. Poderás enviar dados com sucesso; Falaremos mais sobre isso posteriormente.

Um protocolo como o HTTP utiliza um soquete para apenas uma transferência. O cliente envia uma solicitação e depois lê uma resposta. É isso aí. O soquete é descartado. Isso significa que um cliente pode detectar o final da resposta ao receber 0 bytes.

But if you plan to reuse your socket for further transfers, you need to realize that *there is no* EOT (End of Transfer) *on a socket*. I repeat: if a socket `send` or `recv` returns after handling 0 bytes, the connection has been broken. If the connection has *not* been broken, you may wait on a `recv` forever, because the socket will *not* tell you that there's nothing more to read (for now). Now if you think about that a bit, you'll come to realize a fundamental truth of sockets: *messages must either be fixed length* (yuck), *or be delimited* (shrug), *or indicate how long they are* (much better), *or end by shutting down the connection*. The choice is entirely yours, (but some ways are righter than others).

Assuming you don't want to end the connection, the simplest solution is a fixed length message:

```
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)
```

The sending code here is usable for almost any messaging scheme - in Python you send strings, and you can use `len()` to determine its length (even if it has embedded `\0` characters). It's mostly the receiving code that gets more complex. (And in C, it's not much worse, except you can't use `strlen` if the message has embedded `\0`s.)

The easiest enhancement is to make the first character of the message an indicator of message type, and have the type determine the length. Now you have two `recvs` - the first to get (at least) that first character so you can look up the length, and the second in a loop to get the rest. If you decide to go the delimited route, you'll be receiving in some arbitrary chunk size, (4096 or 8192 is frequently a good match for network buffer sizes), and scanning what you've received for a delimiter.

Uma complicação a ter em conta: se o seu protocolo de conversa permitir que várias mensagens sejam enviadas de volta para trás (sem algum tipo de resposta) e você passar `recv` um tamanho arbitrário de bloco, você pode acabar lendo o início de uma mensagem seguinte. Você precisará deixar isso de lado e segurá-lo até que seja necessário.

Prefixing the message with its length (say, as 5 numeric characters) gets more complex, because (believe it or not), you may not get all 5 characters in one `recv`. In playing around, you'll get away with it; but in high network loads, your code will very quickly break unless you use two `recv` loops - the first to determine the length, the second to get the data part of the message. Nasty. This is also when you'll discover that `send` does not always manage to get rid of everything in one pass. And despite having read this, you will eventually get bit by it!

In the interests of space, building your character, (and preserving my competitive position), these enhancements are left as an exercise for the reader. Lets move on to cleaning up.

### 3.1 Dados Binários

It is perfectly possible to send binary data over a socket. The major problem is that not all machines use the same formats for binary data. For example, [network byte order](#) is big-endian, with the most significant byte first, so a 16 bit integer with the value 1 would be the two hex bytes 00 01. However, most common processors (x86/AMD64, ARM, RISC-V), are little-endian, with the least significant byte first - that same 1 would be 01 00.

Socket libraries have calls for converting 16 and 32 bit integers - `ntohl`, `htonl`, `ntohs`, `htons` where “n” means *network* and “h” means *host*, “s” means *short* and “l” means *long*. Where network order is host order, these do nothing, but where the machine is byte-reversed, these swap the bytes around appropriately.

In these days of 64-bit machines, the ASCII representation of binary data is frequently smaller than the binary representation. That's because a surprising amount of the time, most integers have the value 0, or maybe 1. The string "0" would be two bytes, while a full 64-bit integer would be 8. Of course, this doesn't fit well with fixed-length messages. Decisions, decisions.

## 4 Desconectando

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 Quando o Soquete Morre

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 Soquetes não-bloqueantes

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(False)` 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.)

A principal diferença mecânica é que `send`, `recv`, `connect` e `accept` podem retornar sem terem feito nada. Terás (claro) uma série de escolhas. Poderás verificar o código de retorno e os códigos de erro que geralmente nos deixam loucos. Se não acreditas em mim, tente alguma vez. Seu aplicativo vai crescendo, buggy e sugam CPU. Então, vamos ignorar as soluções mortas no cérebro e fazê-lo direito.

Utilize `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.