# Theory: Socket module



20 minutes 0 / 5 problems solved



In the world where the Internet helps to connect the opposite sides of the Earth, it is important to know how this connection is done. The flow of the data in the network has a direction. To allow for this directed flow, we need a start point A and an end point B, so that they, knowing the locations of each other, could open a connection by means of a certain mechanism called **sockets**. Although sockets usually operate "behind the scenes", hidden from our eyes by browsers and applications, right now we'll learn how they help to provide the connection between different devices within the network.

## §1. What is a socket?

Here's a classic real-life analogy to explain what sockets are and how they work. Imagine *an information office* you want to visit (let it be a metaphor of, say, a website or a process with which you establish a connection). It is situated at a particular *address* (hostname). When you come there, you see a lot of *tables* (ports). Some of them are empty, but at the others, there are *consultants* (**server sockets**) who are ready to help you with your question. To get the information you want, *you* (a client socket) need to come to some of such tables, *start a conversation* (establish a connection), *ask your question* (send the request) and *get an answer* (receive the response).

In formal terms, a networking **socket** is an interface that plays an important part in enabling the connection between two processes exchanging data. Specifically, there is a socket at each end of the connection, so it performs as an *endpoint*, and it can send the data to the other end and receive the response, if the socket on the other end sends it. Sockets at the opposite ends are not identical in their functions: one of them is a **server socket**, a listening socket bound to the particular address and waiting for some **client socket** to connect for data exchange. For the client socket to find the server and successfully connect, the client socket must as well be provided with the *address* to which the server socket is bound. This address consists of a *hostname* (IP address or an Internet domain) and a *port number.*

Now let's move from theory to practice and take a look at socket module in Python!

## §2. Creating a client socket

Let's start with an easy one — a client socket. Keep in mind, though, that creating a client socket only makes sense when you already have something to connect to, for example, when you create a server socket and run it yourself or when you simply know the address of some listening socket you need.

First of all, we need to import the module and create our socket.

```
import socket

# creating the socket
client_socket = socket.socket()
```

To provide the socket with the address to connect to, we should create a tuple containing two elements: the first one is the hostname, the second one is the port. Remember, the hostname is always a **string**, while the port is an **integer**. So far, let's take a string '127.0.0.1' as a hostname — this is the address that allows your computer to establish a connection with itself (this address is needed when there is a server running on your computer).

```
hostname = '127.0.0.1'
```

Current topic:

Socket module

Topic depends on:

✗ Domains Stage 4

✗ Creating bytes

✓ Tuple

✓ Declaring a function Stage 1 10

✓ Load module Stage 1 5

✓ Context manager Stage 2

Table of contents:

↑ Socket module

§1. What is a socket?

§2. Creating a client socket

§3. Sending data

§4. Receiving data

§5. Overview

§6. with ... as

§7. Conclusions

Feedback & Comments

As a port, we can use any number in the range from 0 to 65535. However, usually, only numbers starting from 1024 are chosen, since ports from 0 to 1023 are system ones.

```
# let's choose the number 9090 for our port
port = 9090
```

Then we combine these two parts into a single tuple.

```
address = (hostname, port)
```

The next step is to establish a connection to the given address. This can be done with the help of `connect()` method of the socket we've created.

```
client_socket.connect(address)
```

Now let's see what we can do with our socket next.

## §3. Sending data

If the connection is successful, nothing prevents us from finally sending our data to the server socket. `send()` method is what we need for that.

An important note: what you send through your socket must be necessarily in binary format. So, whatever data you want to submit, make sure you've converted it to `bytes` data type first.

```
data = 'Wake up, Neo'
# converting to bytes
data = data.encode()
# sending through socket
client_socket.send(data)
```

Note also that you can't send an empty byte string through a socket. It only happens automatically when you close the connection.

Usually we want not to simply submit our data, but also receive the server's response to it, since it may contain valuable data (for example, if you send some inquiry to the server, the server socket sends you back the results of the search). Let's see how this is done!

## §4. Receiving data

We can receive the response of the server socket with `recv()` method. `recv()` requires a buffer size as an argument — an integer argument specifying the maximum number of bytes to be received at once. The returned response of the server is also always in binary format, so you may want to convert it back to a string.

```
response = client_socket.recv(1024)
# decoding from bytes to string
response = response.decode()
print(response)
```

We can repeat the process of sending-receiving the data if needed. Once we're done, we simply end the connection with `close()` method.

```
client_socket.close()
```

So, we've gone through the main stages of the socket's life. Let's recall them all once again.

## §5. Overview

Here you can take a final look at the whole code.

```
import socket

# creating the socket
client_socket = socket.socket()
hostname = '127.0.0.1'
port = 9090
address = (hostname, port)

# connecting to the server

client_socket.connect(address)



data = 'Wake up, Neo'

# converting to bytes

data = data.encode()



# sending through socket

client_socket.send(data)



# receiving the response

response = client_socket.recv(1024)



# decoding from bytes to string

response = response.decode()

print(response)



client_socket.close()
```

If you run this code just like that, without creating a server socket first, don't expect it to work. You'll get the `ConnectionRefusedError`: this means that connection has failed because we tried to connect to the address that no server socket listens to. This is logical since hostname '127.0.0.1' indicates that we connect to our own computer — and there's no server socket running. To fix the error, you should bind a server socket to the same address and run it.

So, as we see, the structure of a client socket is very simple: connect, send the data, receive the answer, end the discussion by closing the socket. The things are a bit more complicated with server sockets, but that's a story for another topic.

## §6. with ... as

Sockets, just like file objects, can be used as context managers. In practice, this means that we can simplify the process of ending the connection by using the `with` keyword. Let's take a look at the same socket we've been working with, but used in this construction:

```
import socket

# working with a socket as a context manager
with socket.socket() as client_socket:
    hostname = '127.0.0.1'
    port = 9090
    address = (hostname, port)

    client_socket.connect(address)



    data = 'Wake up, Neo'

    data = data.encode()



    client_socket.send(data)



    response = client_socket.recv(1024)



    response = response.decode()

    print(response)
```

As you can see, not much has changed but here we can be sure that the connection will be safely closed and no errors will arise!

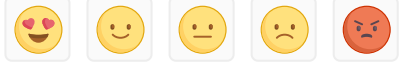
## §7. Conclusions

Let's go over the main points of the topic:

- sockets are endpoints of the connection between two processes
- there are server sockets listening to particular ports, and client sockets that initiate the connection and send the data first
- a socket needs an address to bind or connect to, and it consists of a hostname and a port number
- the data sent through sockets must be `bytes`
- the main steps of the client socket performance involve connection, sending the data, receiving the response and closing the connection

Report a typo

**154** users liked this theory. **2** didn't like it. **What about you?**



Start practicing

Comments (3) Hints (0) Useful links (3) Show discussion