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 which you establish a connection with). 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!

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'
```

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.

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 should be 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 only to submit our data, but also to 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!

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.

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.

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!

Summary

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

Python Socket Library

For socket programming in Python, we use the official built-in Python socket library consisting of functions, constants, and classes that are used to create, manage and work with sockets. Some commonly used functions of this library include:

1. socket(): Creates a new socket.
2. bind(): Associates the socket to a specific address and port.
3. listen(): Starts listening for incoming connections on the socket.

4. accept(): Accepts a connection from a client and returns a new socket for communication.
5. connect(): Establishes a connection to a remote server.
6. send(): Sends data through the socket.
7. recv(): Receives data from the socket.
8. close(): Closes the socket connection.

Creating socket server in Python

Let's start with creating a socket server (Python TCP server, in particular, since it will be working with TCP sockets, as we will see), which will exchange messages with clients. To clarify the terminology, while technically any server is a socket server, since sockets are always used under the hood to initiate network connections, we use the phrase "socket server" because our example explicitly makes use of socket programming.

So, follow the steps below:

Creating python file with some boilerplate

- Create a file named server.py
- Import the socket module in your Python script.

```
import socket
```

- Add a function called run_server. We will be adding most of our code there. When you add your code to the function, don't forget to properly indent it:

```
def run_server():
    # your code will go here
```

Instantiating socket object

As a next step, in run_server, create a socket object using the socket.socket() function.

The first argument (socket.AF_INET) specifies the IP address family for IPv4 (other options include: AF_INET6 for IPv6 family and AF_UNIX for Unix-sockets)

The second argument (socket.SOCK_STREAM) indicates that we are using a TCP socket.

In case of using TCP, the operating system will create a reliable connection with in-order data delivery, error discovery and retransmission, and flow control. You will not have to think about implementing all those details.

There is also an option for specifying a UDP socket: socket.SOCK_DGRAM. This will create a socket which implements all the features of UDP under the hood.

In case you want to go more low-level than that and build your own transport layer protocol on top of the TCP/IP network layer protocol used by sockets, you can use socket.RAW_SOCKET value for the second argument. In this case the operating system will not handle any higher level protocol features for you and you will have to implement all the headers, connection confirmation and retransmission functionalities yourself if you need them. There are also other values that you can read about in the **documentation**.

```
# create a socket object
    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```
OpenAI

Binding server socket to IP address and port

Define the hostname or server IP and port to indicate the address which the server will be reachable from and where it will listen for incoming connections. In this example, the server is listening on the local machine - this is defined by the server_ip variable set to 127.0.0.1 (also called localhost).

The port variable is set to 8000, which is the port number that the server application will be identified by by the operating system (It is recommended to use values above 1023 for your port numbers to avoid collisions with ports used by system processes).

```
server_ip = "127.0.0.1"
    port = 8000
```
OpenAI

Prepare the socket to receive connections by binding it to the IP address and port which we have defined before.

```
# bind the socket to a specific address and port
    server.bind((server_ip, port))
```
OpenAI

Listening for incoming connections

Set up a listening state in the server socket using the listen function to be able to receive incoming client connections.

This function accepts an argument called backlog which specifies the maximum number of queued unaccepted connections. In this example, we use the value 0 for this argument. This means that only a single client can interact with the server. A connection attempt of any client performed while the server is working with another client will be refused.

If you specify a value that is bigger than 0, say 1, it tells the operating system how many clients can be put into the queue before the accept method is called on them.

Once accept is called a client is removed from the queue and is no longer counted towards this limit. This may become clearer once you see further parts of the code, but what this parameter essentially does can be illustrated as follows: once your listening server receives the connection request it will add this client to the queue and proceed to accepting it's request. If before the server was able to internally call accept on the first client, it receives a connection request from a second client, it will push this second client to the same queue provided that there is enough space in it. The size of exactly this queue is controlled by the backlog argument. As soon as the server accepts the first client, this client is removed from the queue and the server starts communicating with it. The second client is still left in the queue, waiting for the server to get free and accept the connection.

If you omit the backlog argument, it will be set to your system's default (under Unix, you can typically view this default in the /proc/sys/net/core/somaxconn file).

```
# listen for incoming connections
    server.listen(0)
    print(f"Listening on {server_ip}:{port}")
```

OpenAI

Accepting incoming connections

Next, wait and accept incoming client connections. The accept method stalls the execution thread until a client connects. Then it returns a tuple pair of (conn, address), where address is a tuple of the client's IP address and port, and conn is a new socket object which shares a connection with the client and can be used to communicate with it.

accept creates a new socket to communicate with the client instead of binding the listening socket (called server in our example) to the client's address and using it for the communication, because the listening socket needs to listen to further connections from other clients, otherwise it would be blocked. Of course, in our case, we only ever handle a single client and refuse all the other connections while doing so, but this will be more relevant once we get to the multithreaded server example.

```
# accept incoming connections
    client_socket, client_address = server.accept()
    print(f"Accepted connection from
{client_address[0]}:{client_address[1]}")
```
OpenAI

Creating communication loop

As soon as a connection with the client has been established (after calling the accept method), we initiate an infinite loop to communicate. In this loop, we perform a call to the recv method of the client_socket object. This method receives the specified number of bytes from the client - in our case 1024.

1024 bytes is just a common convention for the size of the payload, as it's a power of two which is potentially better for optimization purposes than some other arbitrary value. You are free to change this value however you like though.

Since the data received from the client into the request variable is in raw binary form, we transformed it from a sequence of bytes into a string using the decode function.

Then we have an if statement, which breaks out of the communication loop in case we receive a "close" message. This means that as soon as our server gets a "close" string in request, it sends the confirmation back to the client and terminates its connection with it. Otherwise, we print the received message to the console. Confirmation in our case is just sending a "closed" string to the client.

Note that the lower method that we use on the request string in the if statement, simply converts it to lowercase. This way we don't care whether the close string was originally written using uppercase or lowercase characters.

```
# receive data from the client
    while True:
        request = client_socket.recv(1024)
        request = request.decode("utf-8") # convert bytes to
string

        # if we receive "close" from the client, then we break
        # out of the loop and close the conneciton
        if request.lower() == "close":
            # send response to the client which acknowledges
that the
            # connection should be closed and break out of the
loop
```

```
            client_socket.send("closed".encode("utf-8"))
            break

        print(f"Received: {request}")
```
OpenAI

Sending response back to client

Now we should handle the normal response of the server to the client (that is when the client doesn't wish to close the connection). Inside the while loop, right after print(f"Received: {request}"), add the following lines, which will convert a response string ("accepted" in our case) to bytes and send it to the client. This way whenever server receives a message from the client which is not "close", it will send out the "accepted" string in response:

```
response = "accepted".encode("utf-8") # convert string to bytes
# convert and send accept response to the client
client_socket.send(response)
```
OpenAI

Freeing resources

Once we break out from the infinite while loop, the communication with the client is complete, so we close the client socket using the close method to release system resources. We also close the server socket using the same method, which effectively shuts down our server. In a real world scenario, we would of course probably want our server to continue listening to other clients and not shut down after communicating with just a single one, but don't worry, we will get to another example further below.

For now, add the following lines after the infinite while loop:

```
# close connection socket with the client
    client_socket.close()
    print("Connection to client closed")
    # close server socket
    server.close()
```
OpenAI

Note: don't forget to call the run_server function at the end of your server.py file. Simply use the following line of code:

```
run_server()
```
OpenAI

Complete server socket code example

Here is the complete server.py source code:

```
import socket


def run_server():
    # create a socket object
    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    server_ip = "127.0.0.1"
    port = 8000

    # bind the socket to a specific address and port
    server.bind((server_ip, port))
    # listen for incoming connections
    server.listen(0)
    print(f"Listening on {server_ip}:{port}")

    # accept incoming connections
    client_socket, client_address = server.accept()
    print(f"Accepted connection from
{client_address[0]}:{client_address[1]}")

    # receive data from the client
    while True:
        request = client_socket.recv(1024)
        request = request.decode("utf-8") # convert bytes to
string

        # if we receive "close" from the client, then we break
        # out of the loop and close the conneciton
        if request.lower() == "close":
            # send response to the client which acknowledges
that the
            # connection should be closed and break out of the
loop
            client_socket.send("closed".encode("utf-8"))
            break
```

```
        print(f"Received: {request}")

        response = "accepted".encode("utf-8") # convert string
to bytes
        # convert and send accept response to the client
        client_socket.send(response)

    # close connection socket with the client
    client_socket.close()
    print("Connection to client closed")
    # close server socket
    server.close()


run_server()
```

OpenAI

Note that in order not to convolute and complicate this basic example, we omitted the error handling. You would of course want to add try-except blocks and make sure that you always close the sockets in the finally clause. Continue reading and we will see a more advanced example.

Complete client socket code example

Here is the complete client.py code:

```
import socket
```

```
def run_client():
    # create a socket object
    client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    server_ip = "127.0.0.1"  # replace with the server's IP
address
    server_port = 8000  # replace with the server's port number
    # establish connection with server
    client.connect((server_ip, server_port))

    while True:
        # input message and send it to the server
        msg = input("Enter message: ")
        client.send(msg.encode("utf-8")[:1024])

        # receive message from the server
        response = client.recv(1024)
        response = response.decode("utf-8")

        # if server sent us "closed" in the payload, we break
out of the loop and close our socket
        if response.lower() == "closed":
            break

        print(f"Received: {response}")

    # close client socket (connection to the server)
    client.close()
    print("Connection to server closed")

run_client()
```

Test your client and server

To test the the server and client implementation that we wrote above, perform the following:

- Open two terminal windows simultaneously.

- In one terminal window, navigate to the directory where the server.py file is located and run the following command to start the server:

```
python server.py
```

This will bind the server socket to the localhost address (127.0.0.1) on port 8000 and start listening for incoming connections.

- In the other terminal, navigate to the directory where the client.py file is located and run the following command to start the client:

```
python client.py
```

This will prompt for user input. You can then type in your message and press Enter. This will transfer your input to the server and display it in its terminal window. The server will send its response to the client and the latter will ask you for the input again. This will continue until you send the ”close” string to the server.