# Module 24: Networking with Python

## 24.1: Computer Networks

Computer networks have become an integral part of our lives, mainly in the form of the Internet and Voice Over IP (Internet Protocol). A complicated eco-system of layers and protocols determines how digital messages between two end-points are created, sent, received, and interpreted. A basic strategy for the success of the technology was a layered design. In practice, the task of defining networking is divided into the following layers:

- Physical layer: how is information sent over a connection such as an ethernet cable or between two wireless access points. This layer is concerned about defining the signal transmission over a physical medium.
- Link layer: how is data exchanged between two directly connected nodes such as a computer and a wireless access point or a phone and a headset using Bluetooth.
- Network layer: how is data exchanged between two nodes when the connection goes through intermediate networks.
- Transport layer: how do nodes handle the data exchange of data between two connected devices. The transport layer defines the view of applications that need to use networking.
- Application layer: How do different applications communicate over a network connection. For example, http and sftp are protocols that belong to the application layer.

This division between layers allows the definition of protocols that are independent. For example, Transmission Control Protocol (TCP), part of the transport layer, does not need to know how messages on a sub-link might be exchanged as electronic signals and vice versa. Networking with Python is built on top of the transport layer. Currently and for the foreseeable future, this is dominated by the TCP and the User Datagram Protocol (UDP). They provide for data exchange between two applications residing in two different or in a single machine. As such, they have addresses, consisting of two parts:

1. The internet address. The internet address is often still an IPv4 (Internet Protocol Version4) address of 32 bits, written as a quadruple of integers between 0 and 255. For example, the address 127.0.0.1 is known as localhost and tells the network that a message with this address is directed to the originating entity. This address is used if we use the network to send messages to another application on the same machine. The alternative to IPv4 network routing is IPv6. IPv6 addresses have 128 bits. They are usually given as an octuple of 16 bits, where each of the eight components is written as a hexadecimal number with four digits. The loopback address is 0000:0000:0000:0000:0000:0000:0000:0001, which is abbreviated to 0:0:0:0:0:0:0:1 or even more to ::1.

2. The port number. Port numbers are used to address individual applications using one of the two dominant transport level protocols, namely TCP and UDP. Both protocols use port numbers between 0 and 65535 ($2^{16} - 1$). The range of port numbers is divided into three regions. Port numbers between 0 and 1024 are *well-known* port numbers and are assigned to certain protocols. For example, port 115 for TCP belongs to the Simple File Transfer Protocol. Numbers between 1024 and 49151 are registered ports, meaning that by convention, a certain type of application will always use this port number, though sometimes more than one application is using a single port number. While using a well-known port number is a recipe for disaster as the OS is likely to listen to it, misinterpret a message sent to such a port and crash or complain, using a reserved port number rarely runs into difficulties since your machine is not likely to run the application that has reserved this port. Numbers above 49151 are ephemeral and free to grab for any program.

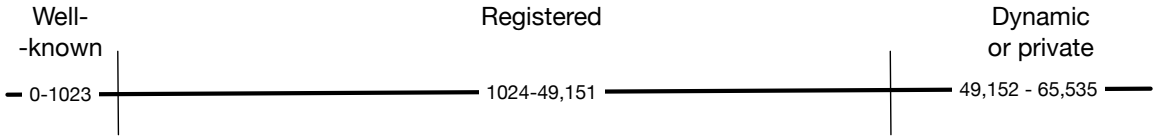


*Figure 1: Division of port numbers*

To communicate with two different applications, you have the choice between UDP and TCP. They offer different performance and operating guarantees. UDP messages might be received out of order or not at all, whereas TCP messages will arrive in order and are guaranteed to arrive, since a corrupted or otherwise lost message will be resent.

# 24.2: Sockets

Python networking is built on Berkeley sockets, originally developed for UNIX 4.2BSD in 1983. Berkeley sockets are still used for Unix systems, whereas Windows OS has Winsock, offering also the same functionality. You can look at a socket as you can look at a file handler. You open a file, then you can either get data from the file through read operations or you can write to the file through write operations. A socket is similar, but it does not have the capability to move the cursor within the file. When you are done with the file, you close it.

A socket supports the following primitives:

SOCKET: Create a new communication end point.
BIND: Attach a local address to the socket
LISTEN: Put the socket into listening mode, where the socket can accept an incoming connection.
ACCEPT: Blocks the execution of the socket program until a connection from outside is attempted. It returns then a new socket.

CONNECT: Actively attempts to connect to a (hopefully listening) socket.
SEND: Send some data outwards.
RECEIVE: Receive some data from a connection.
CLOSE: Release the connection.

You can look at your active sockets using the `netstat` command available on UNIX (including MacOS) and Windows machines.

If you try to establish a connection to another machine, you will have to deal with defensive measures. Firewalls filter out connection attempts that have not been deemed to be safe. Usually, this includes any attempt to open up a TCP or UDP-connection from an outside machine to your machine. If you want to communicate between two different machines, you will have to change the firewall rules temporarily (or even disable the firewall, which would however leave your machine very vulnerable).

Python implements Berkeley sockets in its `socket` module.

# 24.3: Python Networking

We explain Python networking by building a client-server system where the client sends compute requests to the server. To avoid firewall problems, we use the same machine for server and client. The server creates a listening socket. When a client approaches this socket, a new socket is created and passed to the server that now allows communication from and to the client. In contrast, the client connects directly to the server's socket and uses the same socket it used for establishing the connection in order to communicate with the server. We build a "compute-server" that allows a client to connect, submit two numbers, and receive the product back.

## 24.3.1: Server Program

We create a listening socket on local host (or some other computer, whose network address you know and that allows incoming TCP-requests) using IPv4 (specified as `AF_INET)` and TCP. We also pick a port number among the ephemeral port numbers, assuming that this socket is not busy. In order to avoid having to close the socket, we use the `with`-paradigm. Once the socket is created, we bind it to the chosen host and port, and set it into listening mode.

```
import socket

HOST = '127.0.0.1'  #Loopback interface
PORT = 65431  #Silly port

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
```

```
    s.bind((HOST, PORT))
    s.listen()
```

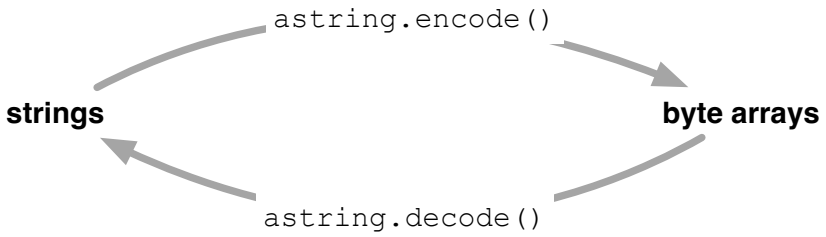


*Figure 2: Encoding and decoding to move data from / to strings to / from byte arrays.*

When a client socket connects to the socket, we accept it, writing down its address. This gives a new socket, with a different port number.

```
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen()
    conn, addr = s.accept()
    print('Connection from:', addr)
```

Now that a connection has been established, we want to receive data. This being done by invoking the `recv` method in the `connection` class. We enter an infinite loop, but we break out of it, if we do not receive any data in a message. As an exercise, you can modify this behavior so that sending a stop message will terminate the server. The data is sent over in binary, with the maximum number of bytes specified in `recv`. We use the string `decode` (and at the client side) the string `encode` methods to move between strings and byte-arrays, Figure 2. After the client has extracted the two numbers, it calculates their product, makes the result into a byte-array, and then uses `send/sendall` to send the result to the client. There is no guarantee that a complete message is being sent, so production code needs to handle this possibility.

```
import socket
HOST = '127.0.0.1'  #Loopback interface
PORT = 54321  #Silly port

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen()
    conn, addr = s.accept()
    print('connection from', conn, addr)
    while True:
        data = conn.recv(256)
        if not data:
            break
```

```
            mystring = data.decode('UTF-8')
            args = mystring.split()
            product = str(int(args[0])*int(args[1]))
            tosend = product.encode('UTF-8')
            conn.sendall(tosend)
        conn.close()
```

### 24.3.2: Python Client

The client connects directly to the server. Thus, the server needs to run first, because otherwise the connection will fail as there is no socket to connect to. We then enter an infinite loop, where we request two numbers from the user interactively. We combine the two numbers (hopefully they are numbers) into a single string and send it to the server. Afterwards, we wait to receive an answer, which we decode and print out.

```
import socket

HOST = '127.0.0.1'  #Loopback interface
PORT = 54321  #Silly port

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((HOST, PORT))
    while True:
        num1 = input('Number 1: ')
        num2 = input('Number 2: ')
        if not num1 or not num2:
            break
        data = num1 + ' ' + num2
        tosend = data.encode('UTF-8')
        s.sendall(tosend)
        result = s.recv(256)
        print(result.decode('UTF-8'))
    s.close()
```

The socket close statement at the end is not necessary, but in this case, it is better to be safe than to be sorry. If you use up too many ports, your computer might become unresponsive.

### 24.3.3: Running the Client-Server

You cannot run both the client and the server simultaneously using IDLE. Open up two terminals (powershells) and navigate to the directory where the server / client is located. Then start the server

```
python3 server.py
```

Nothing will happen, but you can see from the absence of the terminal prompt that the program is running. You might get an error message that the socket address is in use, in which case you can either restart your computer (so that the application which used this socket is no longer running) or change your programs to use a different, but still ephemeral port number. Then you go to the other terminal and start the client. Almost simultaneously, the first terminal will show a connection, and you can verify that it comes from a different port, and the second terminal will show the prompt, asking you for a number, Figure 3. After entering two numbers, the client will send a message to the server, who sends the answer that is then displayed.

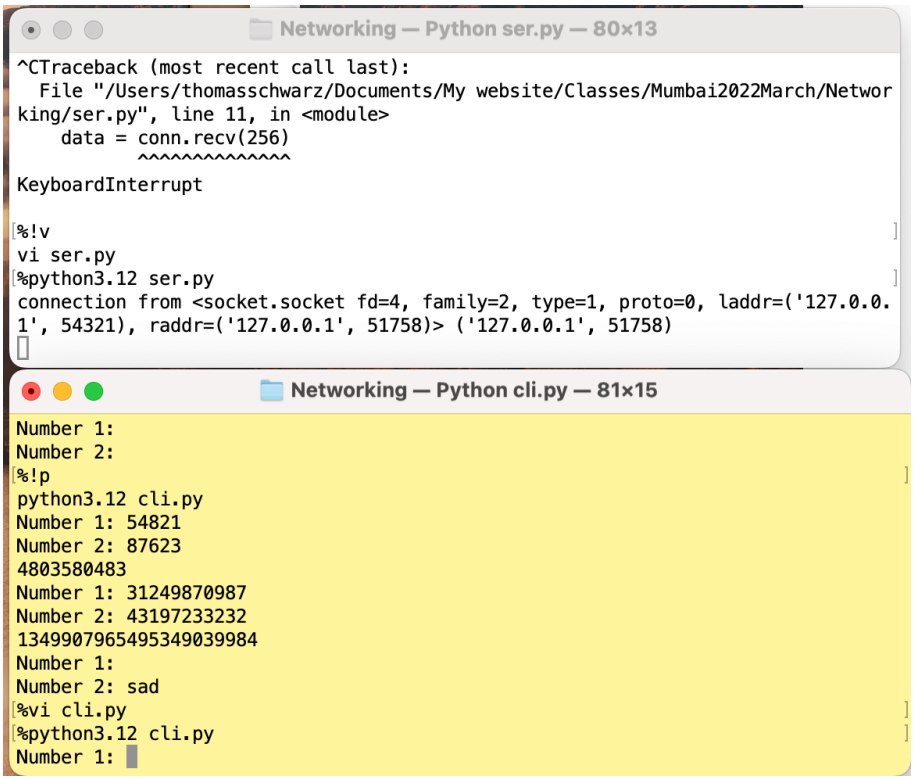


*Figure 3: Two terminals with our server/client application*

# 24.3.4 Further Steps

Our simple client-server shows you how easy Python network programming can be. There is more to be learned, such as how to handle multiple connections, but this would us lead to a discussion of parallelism in Python that would be beyond the scope of this text-book. Even getting Python to work between two different machines can involve quite a bit of system administration, since receiving TCP requests from someone else and responding to it puts a system in jeopardy. Default configurations of individual systems and networks prevent this. If you know the network administrator or if you can build your own network, and if you have administrative rights on your systems then you should know how to put your system back onto a secure footing. If you know how to program firewall rules, you might also be able to keep your systems secure while you are

trying out network programming with Python. You might need to read the Python manuals if you systems have IPv6 addresses.