B. Steenis – Networks 2

Session 5 : Sockets Python Lab - Warning : there is a mandatory report to produce about this lab.

1. Python
2. Sockets
3. Simple application sockets with python + wireshark

Why Python ?

Python is a high-level, general-purpose programming language. Its design philosophy emphasizes code readability with the use of significant indentation.

Python is dynamically typed and garbage-collected. It supports multiple programming paradigms, including structured (particularly procedural), object-oriented and functional programming. It is often described as a "batteries included" language due to its comprehensive standard library.

(wikipedia)

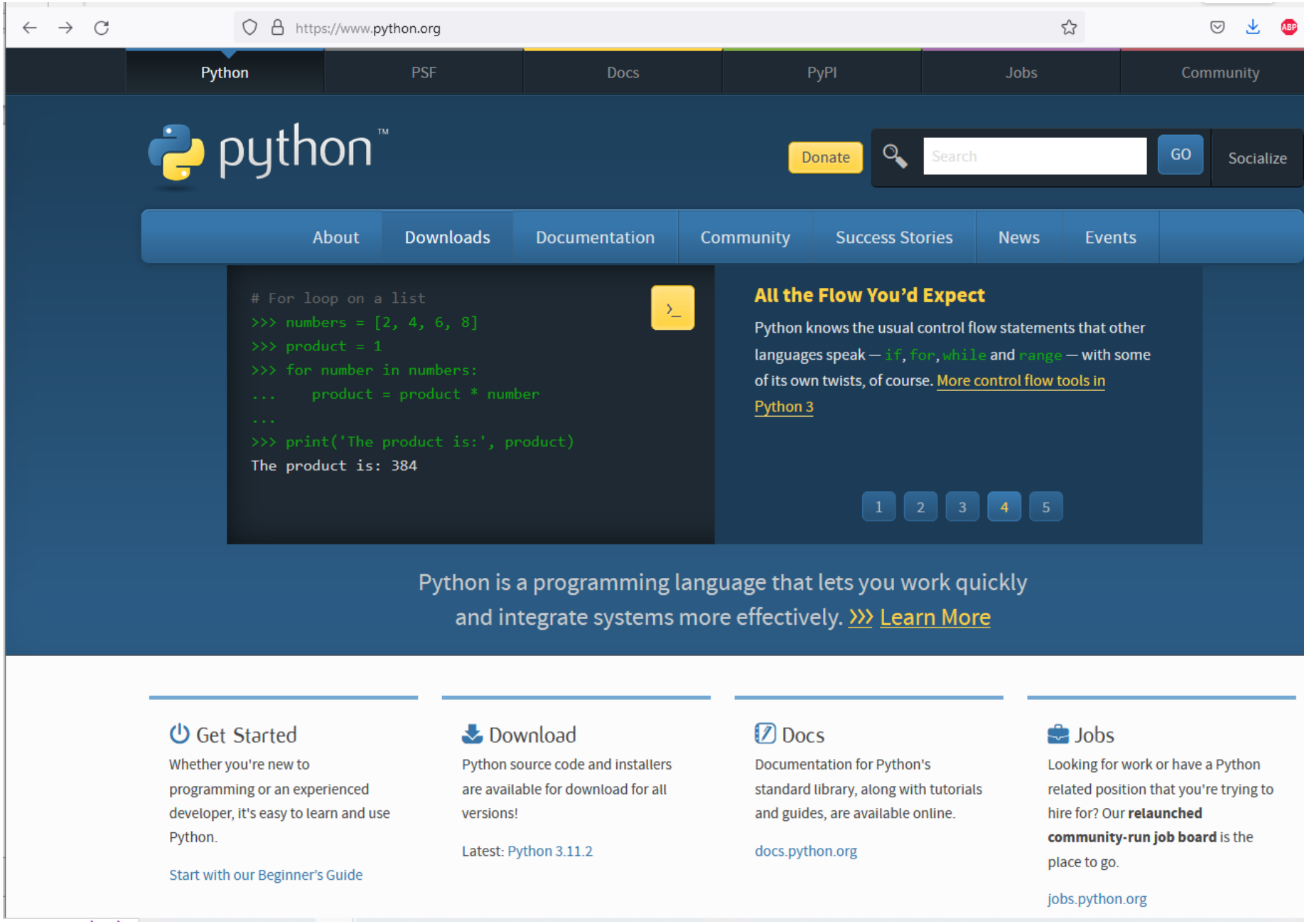
Advantages

- Python is easy to use, easy to learn. Extremely fast learning curve. Extensive help available online.
- Source code is extremely readable, due to indentation implementing programming structures like loops or tests
- Requires no IDE (Integrated development environment)
- Usually interpreted (not compiled) but compilers do exist
- Source code can be view as a script. Can be executed line per line in commad line mode
- Huge community of developers
- Hundreds of available libraries covering almlost everything
- Old langage (first appeared 1991, but recently became popular)
- Mostly used in academic and education world, and for development/testing

The only disadvantage is that execution is not as fast as compiled C/C++ ,
and if you try to compile, Python compilers are generally not efficient (producing huge executable files)

Windows : Install from website : python.org
Linux : already preinstalled on most linux distrbutions.

Python is command-line. **So, you have to add the directory where you installed it in the system PATH environment.**


https://www.python.org
Python
PSF
Docs
PyPI
Jobs
Community
python™
Donate
Search
GO
Socialize
About
Downloads
Documentation
Community
Success Stories
News
Events
# For loop on a list
>>> numbers = [2, 4, 6, 8]
>>> product = 1
>>> for number in numbers:
...   product = product * number
...
>>> print('The product is:', product)
The product is: 384
All the Flow You'd Expect
Python knows the usual control flow statements that other languages speak — if, for, while and range — with some of its own twists, of course. More control flow tools in Python 3
1
2
3
4
5
Python is a programming language that lets you work quickly and integrate systems more effectively. >>> Learn More
Get Started
Whether you're new to programming or an experienced developer, it's easy to learn and use Python.
Start with our Beginner's Guide
Download
Python source code and installers are available for download for all versions!
Latest: Python 3.11.2
Docs
Documentation for Python's standard library, along with tutorials and guides, are available online.
docs.python.org
Jobs
Looking for work or have a Python related position that you're trying to hire for? Our relaunched community-run job board is the place to go.
jobs.python.org

First test …
On Windows prompt, type
> python
On Linux prompt, type
$ python3

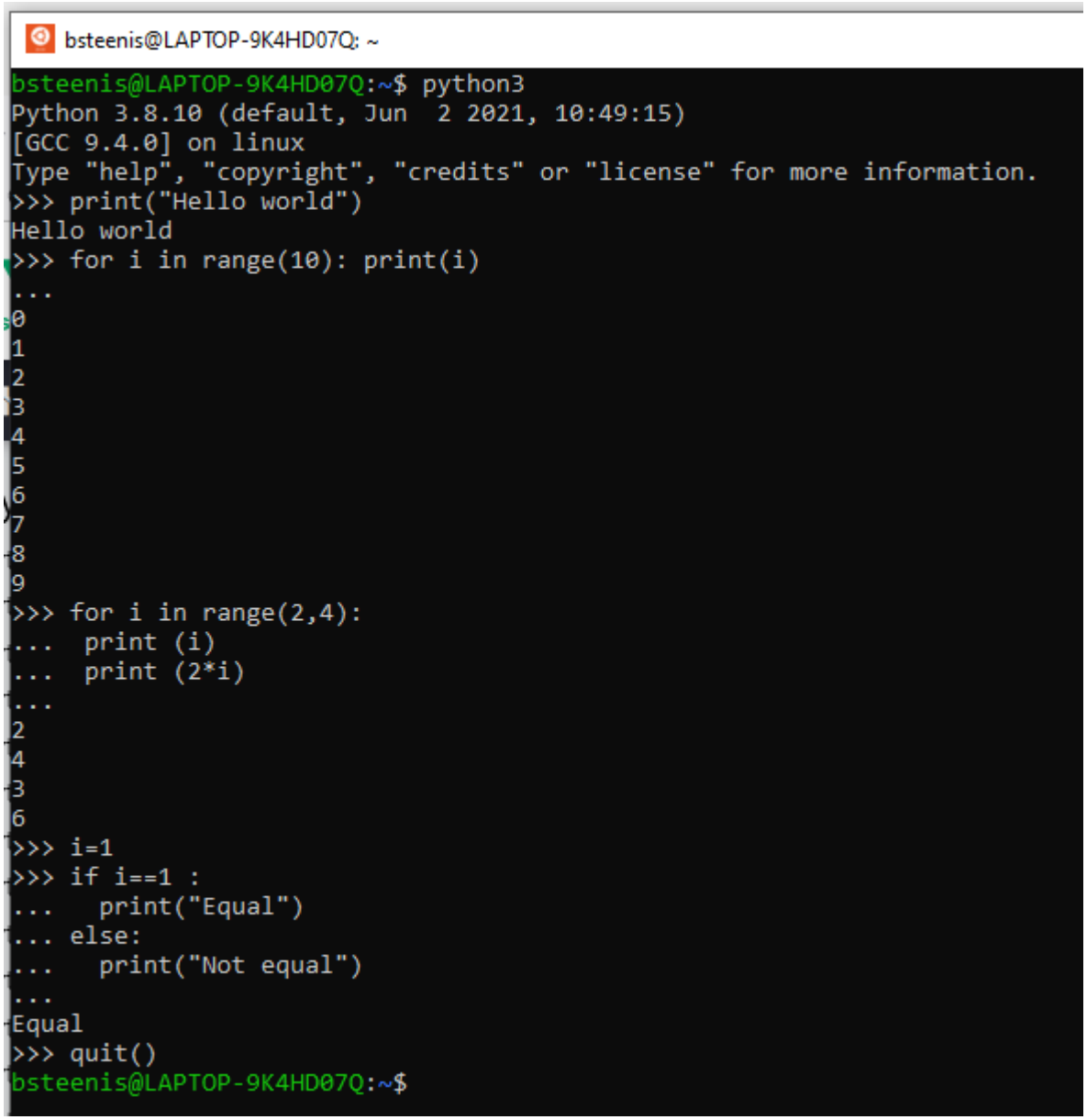
Python prompt is
>>>

On python prompt, you can type any python code. It is immediately executed …
```
>>> print ("Hello world")
```
There is no termination charecter like « ; »

You can also type a loop structure
```
>>> for i in range(10):  print(i)
```
Please remark there is a « : » after the for loop statement. It is the same after a test statement
Second remark, after typing this line you get another prompt, this time is triple point
...
This prompt indicates the statement is incomplete.
If you typed something the same line then the for/while/if … then you must type a blank line (directly with a cariage return), which is the case here
Otherwise you can type the content of the loop or test structure USING INDENTATION
Example :
```
>>> for i in range(2,4):
...     print(i)
...     print(2*i)
```
Then always terminate with a blank line

Example of a test structure
```
>>> i=1
>>> if i==1:
...     print("Equal to 1")
... else:
...     print("Not equal to 1")
```
Remark the test structure, and the indentation. You can of course type multiple lines between if and else and else and the final line

To exit python prompt type
```
>>> quit()
```



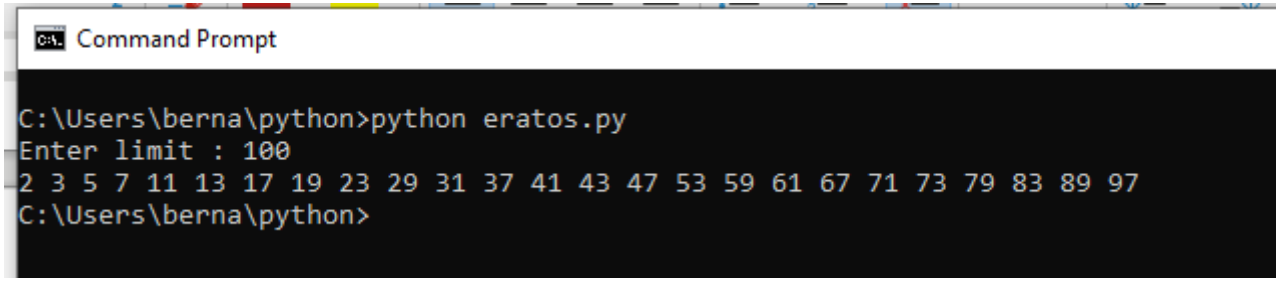
This is an example on a Linux shell. Same in Windows shell.

Of course the source code can be in a text file (source file). You can edit the source with your preferred text editor like Notepad in Windows …

Insert this simple code in a text file named, for example, « eratos.py »
Note that usual extension for python sources is py …

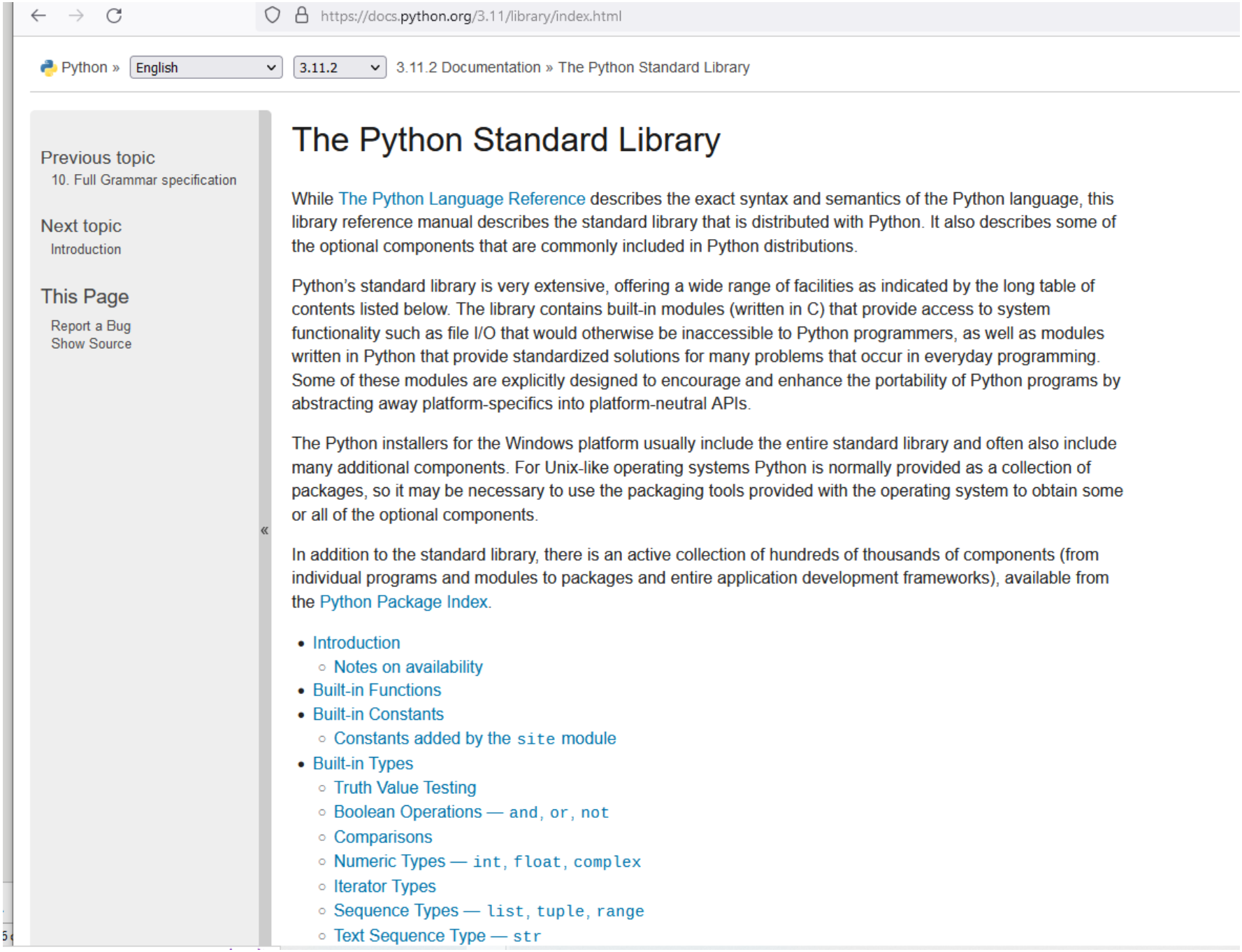
```
#test python : sieve of Eratosthenes
import math
lim=input("Enter limit : ")
lim=int(lim)+1
a=[]
for i in range(lim): a.append(1)
a[1]=0
for i in range(2,1+int(math.sqrt(lim))):
  if a[i]==1:
    for j in range(2*i,lim,i):
       a[j]=0
for i in range(1,lim):
  if a[i]==1: print(i,end=" ")
```

Then on Windos or Linux prompt, just type
> python eratos.py
to execute this source …

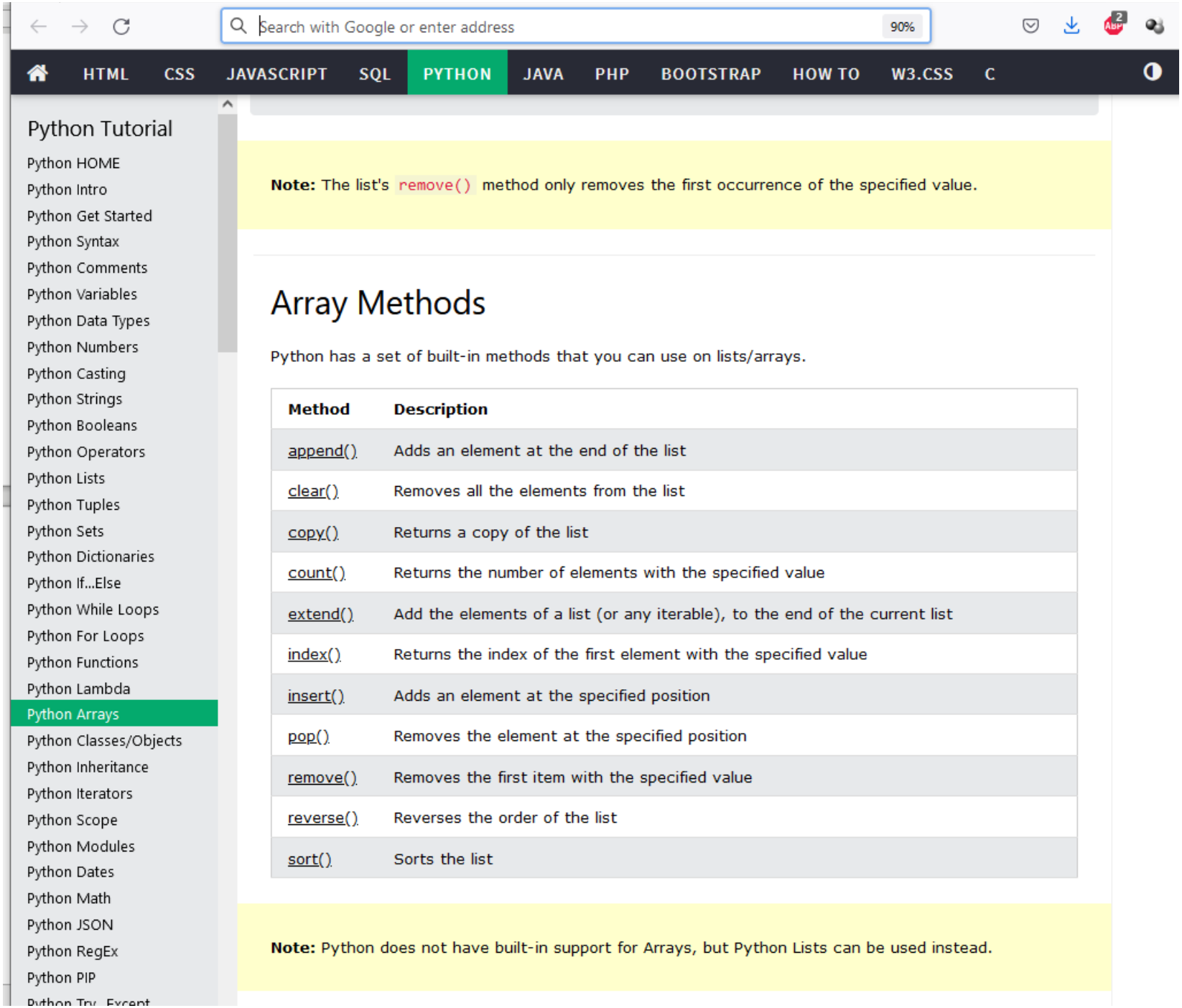


This is Eratosthene's algorithm to get the list of prime numbers to a given limit. The interest of this small piece of code is that it gives a extensive viewpoint of all programing features …
- Against, be careful about indentation. This is really a key of Python sources.
- # is a comment
- import is a library declaration. All libraries should be installed. « math » library is present by default. It is requested for square root sqrt() function
- no variable declaration, they are implicitly declared at first assignation
- input() function to get input from stdin (here input prompt)
- int() is a conversion function to integer
- look at array or structure construction … [] is void array, then append method is applied to add elements
- for / range loop and tests

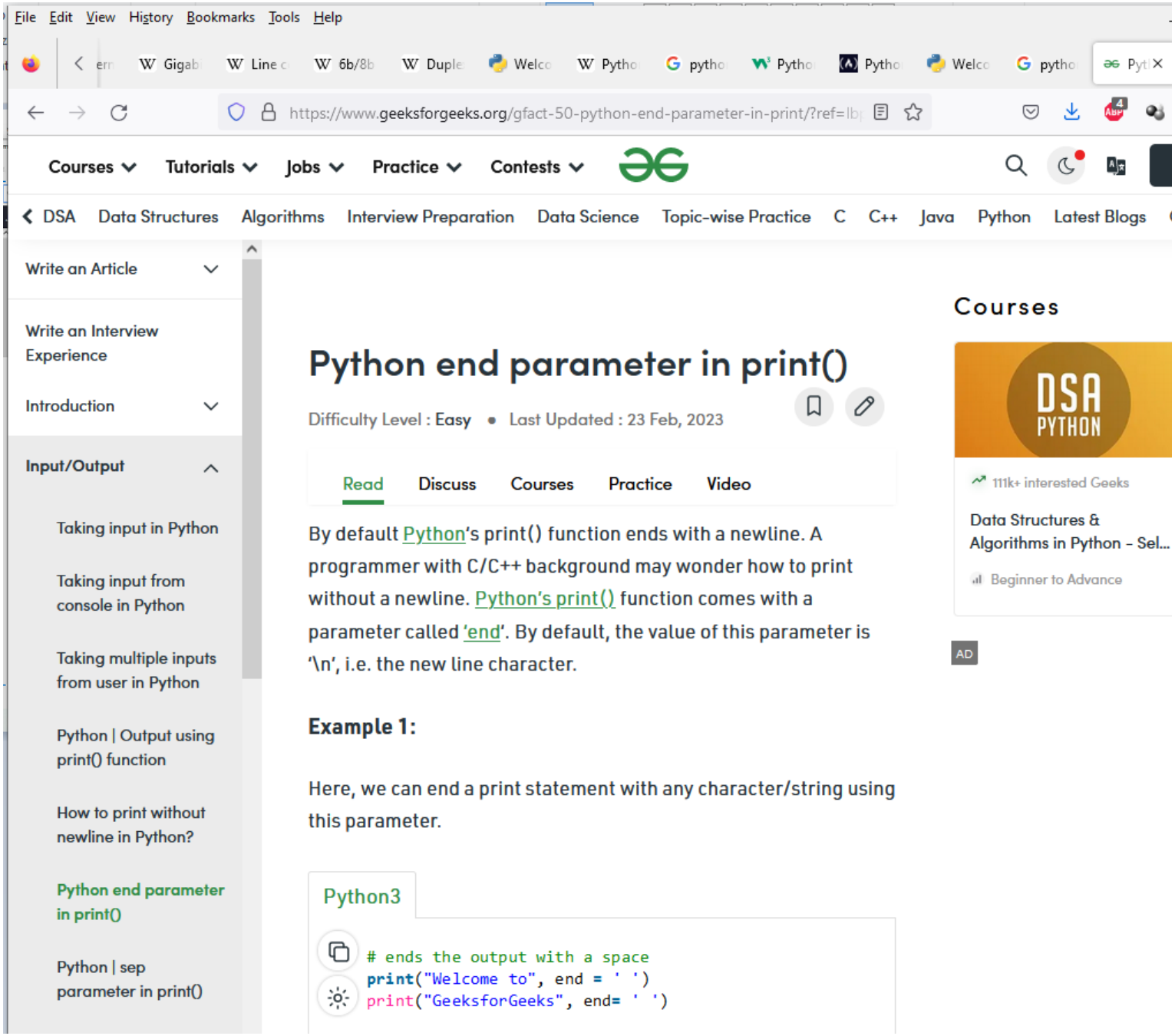
There is a lot of help available online. Never hesitate to type python and a feature/function/command in search engine, like « python while loop »
For core python : Official help from **docs.python.org**



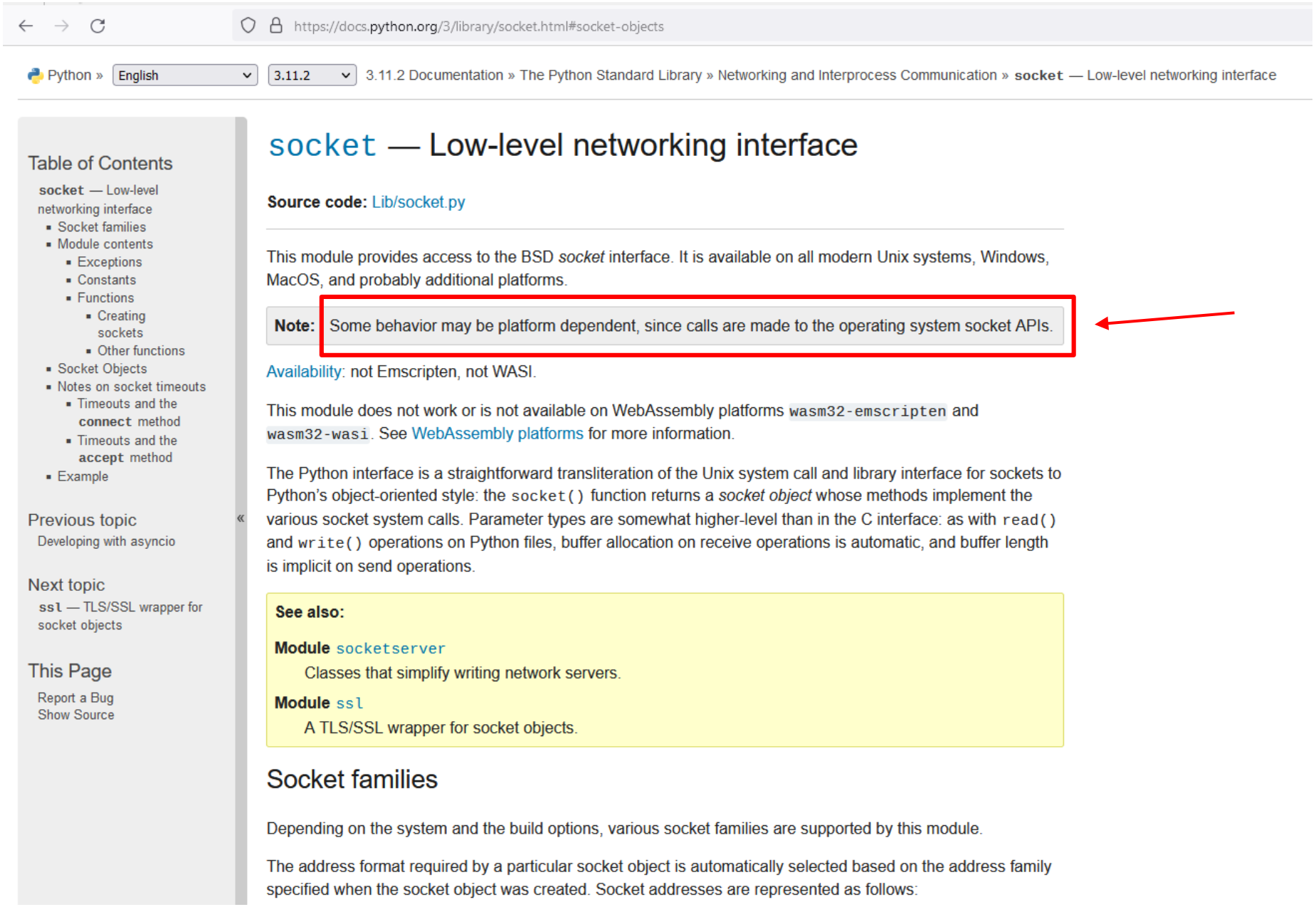
An alternative intersting help source is on **www.w3schools.com/python/**



Or ... **www.geeksforgeeks.org/python-programming-language/**



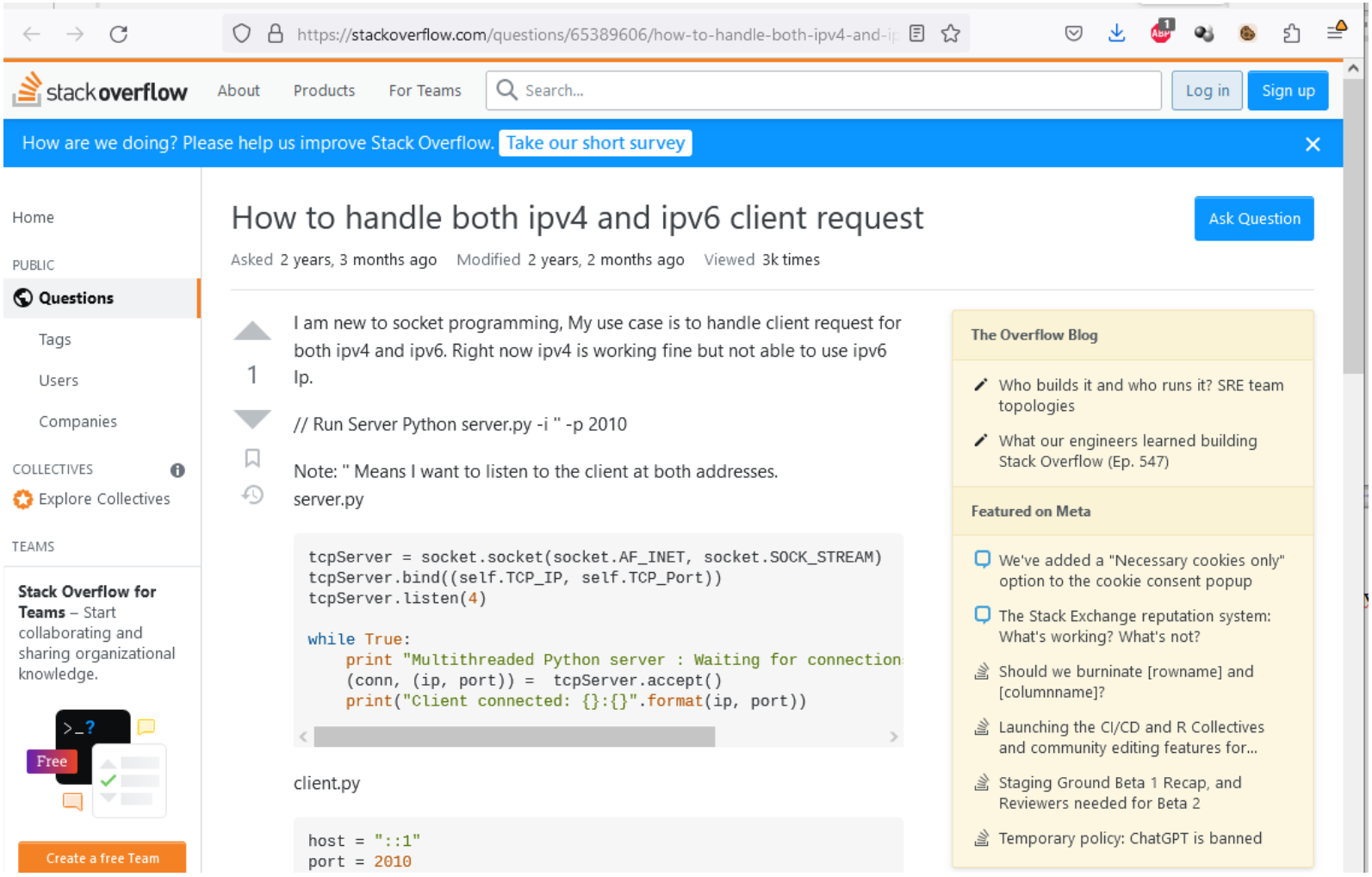
This list of help/tutorial websites is of course not complete. There are really a lot.
For socket progrmmaing we'll have to use the « socket » library which is standard in python.
For external libraries, developers often provide their own online help pages.



About the socket library, one important issue is that the behaviour is strongly platform dependent, and rely on OS APIs
In the case of Windows, only TCP and UDP transport protocols are accepted, and it seems (almost) impossible to access directly to the IP network layer or even below to the Link layer without going through the transport layer.

Otherwise to conclude the huge list of available online help with Python, forums can sometimes be very helpful. If you face a big problem and don't see the solution, think that someone already faced the same problem before you …

The **stackoverflow.com** forum is a gold mine.

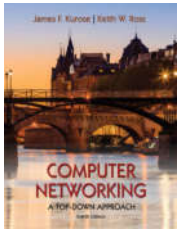


Application Layer

- Principles of network applications
- Web and HTTP
- E-mail, SMTP, IMAP
- The Domain Name System: DNS
- P2P applications
- video streaming, CDNs
- socket programming with UDP and TCP

COMPSCI 453 Computer Networks
Professor Jim Kurose
College of Information and Computer Sciences
University of Massachusetts

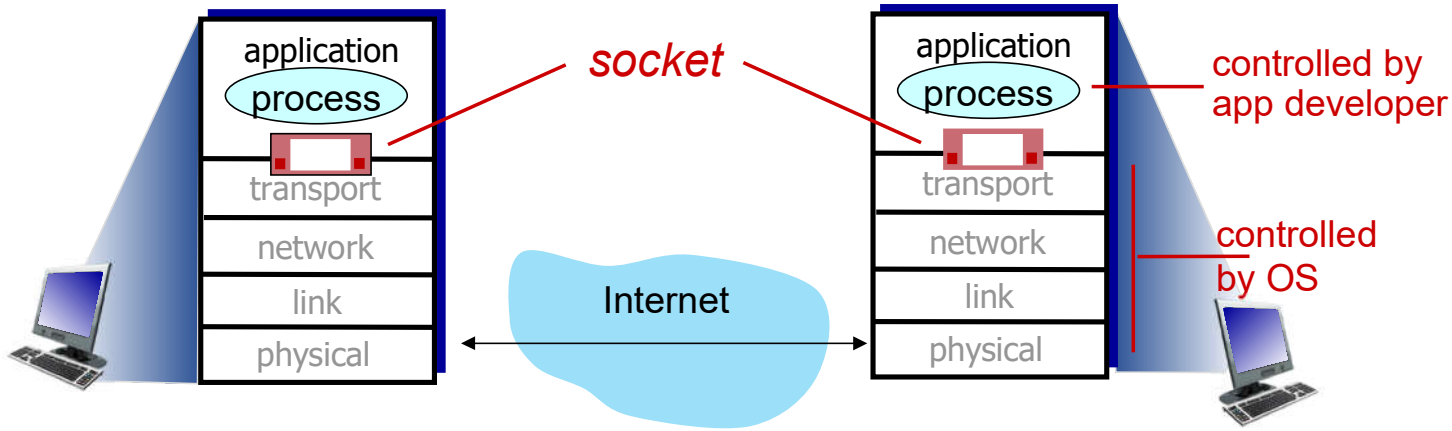
Class textbook:
Computer Networking: A Top-Down Approach (8th ed.)
J.F. Kurose, K.W. Ross
Pearson, 2020
http://gaia.cs.umass.edu/kurose_ross



Socket programming

goal: learn how to build client/server applications that communicate using sockets

socket: door between application process and end-end-transport protocol



Socket programming

Two socket types for two transport services:

- *UDP:* unreliable datagram
- *TCP:* reliable, byte stream-oriented

Application Example:

1. client reads a line of characters (data) from its keyboard and sends data to server
2. server receives the data and converts characters to uppercase
3. server sends modified data to client
4. client receives modified data and displays line on its screen

Socket programming with UDP

UDP: no "connection" between client and server:

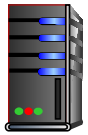
- no handshaking before sending data
- sender explicitly attaches IP destination address and port # to each packet
- receiver extracts sender IP address and port# from received packet

UDP: transmitted data may be lost or received out-of-order

Application viewpoint:

- UDP provides *unreliable* transfer of groups of bytes ("datagrams") between client and server processes

Client/server socket interaction: UDP



server (running on serverIP)

create socket, port= x:
serverSocket =
socket(AF_INET,SOCK_DGRAM)

read datagram from
serverSocket

write reply to
serverSocket
specifying
client address,
port number

client

create socket:
clientSocket =
socket(AF_INET,SOCK_DGRAM)

Create datagram with serverIP address
And port=x; send datagram via
clientSocket

read datagram from
clientSocket

close
clientSocket

Example app: UDP client

Python UDPClient

include Python's socket library →
```
from socket import *
```
```
serverName = 'hostname'
serverPort = 12000
```
create UDP socket for server →
```
clientSocket = socket(AF_INET,
                      SOCK_DGRAM)
```
get user keyboard input →
```
message = raw_input('Input lowercase sentence:')
```
attach server name, port to message; send into socket →
```
clientSocket.sendto(message.encode(),
                    (serverName, serverPort))
```
read reply characters from socket into string →
```
modifiedMessage, serverAddress =
                    clientSocket.recvfrom(2048)
```
print out received string and close socket →
```
print modifiedMessage.decode()
clientSocket.close()
```

Example app: UDP server

Python UDPServer

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET, SOCK_DGRAM)
serverSocket.bind(('', serverPort))
print ("The server is ready to receive")
while True:
    message, clientAddress = serverSocket.recvfrom(2048)
    modifiedMessage = message.decode().upper()
    serverSocket.sendto(modifiedMessage.encode(),
                        clientAddress)
```

create UDP socket →

bind socket to local port number 12000 →

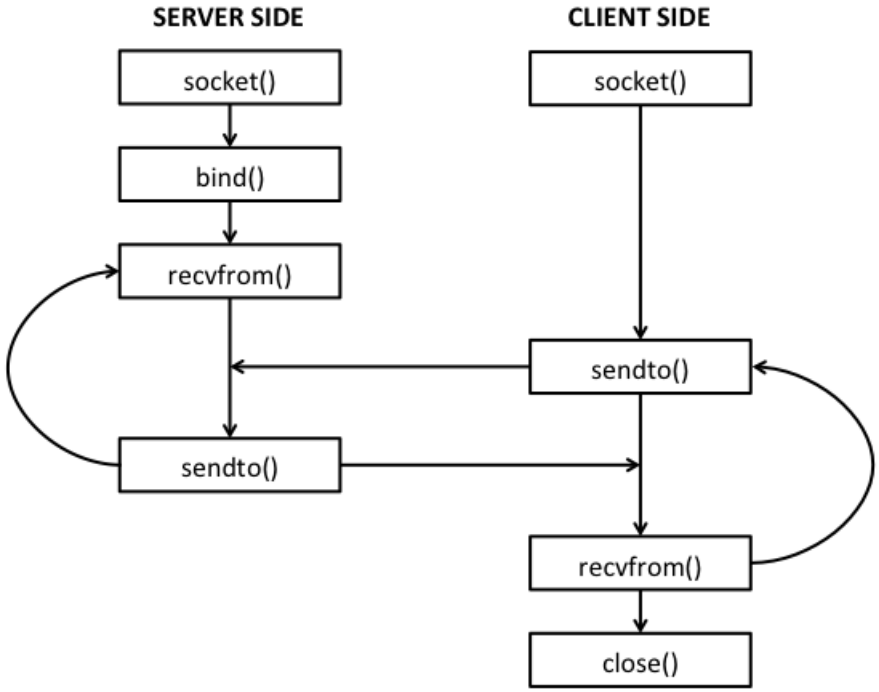
loop forever →

Read from UDP socket into message, getting client's address (client IP and port) →

send upper case string back to this client →

```
# place this code in file client_udp_simple.py :
import socket
host = input(" host -> ")
port = int(input(" port -> "))   # socket server port number
server = (host,port)
client_socket = socket.socket(family=socket.AF_INET,type=socket.SOCK_DGRAM)
message = input(" -> ")  # take input
while message.lower().strip() != 'bye':
    client_socket.sendto(message.encode(),server)
    data,addr = client_socket.recvfrom(1024)
    data=data.decode()  # receive response
    print('Received from server: ')
    print(data)  # show in terminal
    message = input(" -> ")  # again take input
client_socket.close()  # close the connection
```

```
# place this code in file server_udp_simple.py :
import socket
host = input(" host -> ")
port = int(input(" port -> "))
s = socket.socket(family=socket.AF_INET,type=socket.SOCK_DGRAM)
s.bind((host, port))
print("Server Started")
while True:
    data, address = s.recvfrom(1024)
    data = data.decode('utf-8')
    if not data: break
    print("Message from: " + str(address))
    print("Received : ")
    print(data)
    data = data.upper()
    print("Sending: ")
    print(data)
    s.sendto(data.encode('utf-8'),address)
s.close()
```



(https://www.it.uu.se/education/course/homepage/dsp/vt19/modules/module-2/sockets/)

socket.socket (family=socket.AF_INET, type=socket.SOCK_DGRAM)
« socket dot something » is because the function or the constant is defined in the socket library.

socket() creates a socket with two main parameters :
family is the IP protocol, AF_INET for IPv4, AF_INET6 for IPv6. AF_INET (IPv4) is by default and can be omitted
type is the Transport protocol, SOCK_DGRAM for UDP, SOCK_STREAM for TCP. SOCK_STREAM (TCP) is by default and can be omitted.
There are other parameters and many other options defined in the online documentation, however only these combinations work in Windows API.
socket() creates the socket with the IP address of the machine and a randomly generated port. This is convenient for client side.

bind(address,port) makes association between the socket and a given address and port. It is necessary to specify at least a listening port on server side. Address can be left blank, in that case server is listening on all interfaces, or if an address is specified, then server is only listening to the associated interface.

sendto(address,port) sends a packet to the specified destination (address and port)

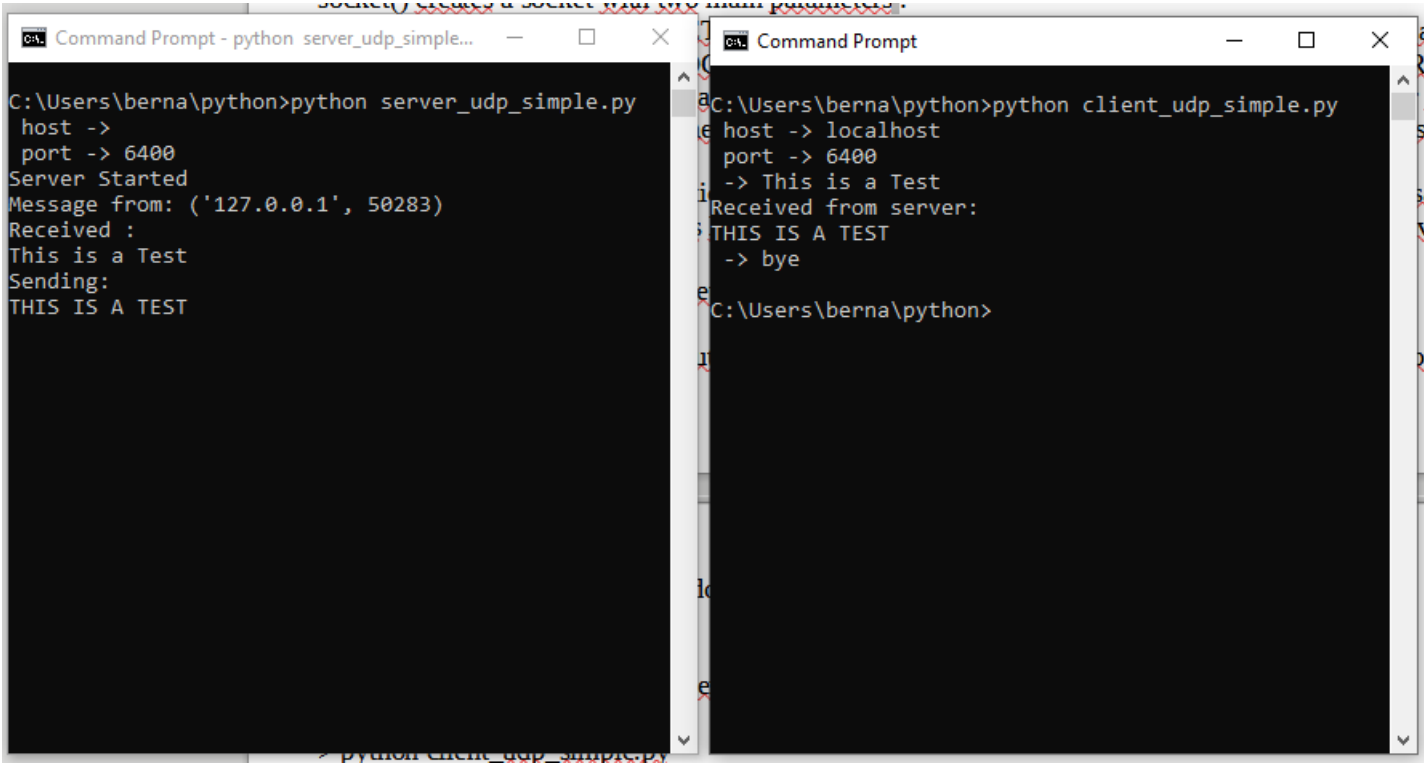
recvfrom() waits and blocks execution until reception of a packet, data and sending address are both returned by the function. Parameter is buffer size.

Open two command shell in Windows, and lauch in the first shell
> python server_udp_simple.py
On host input, leave blank
On port input type any port number you wish, for example 6000
In the second shell, launch
> python client_udp_simple.py
On host input, type localhost
On port input, type the same port number as in the server program.

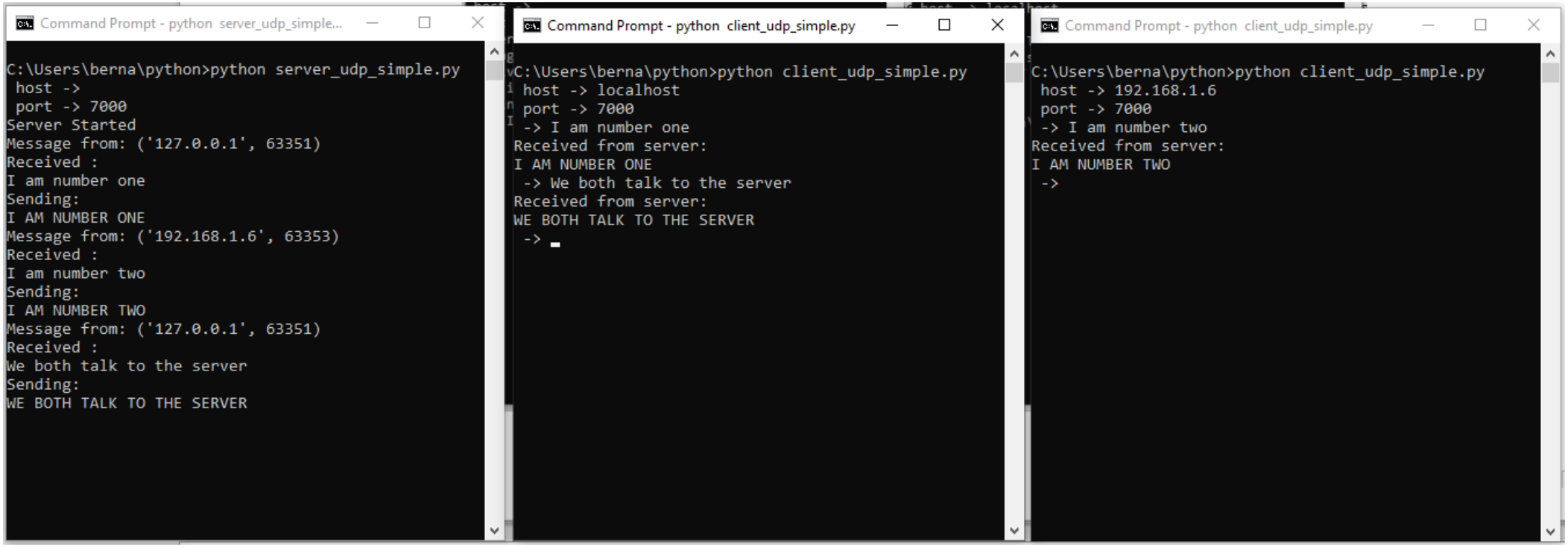
Please remark that an address (host) may be a name, not necessary an IP address. Then the Windows API will perform the appropriate DNS translation.

On prompt in the client program, type any text you wish, it will be transmitted to the server. Upon reception, the server will upper case all characters and send them back to the client. That is all that our server does … echoing in uppercase (this is just how it is coded inn the server, and of course this is just for testing:)

If you type « bye » in the client window, it will close the connection on its side (this is just how it is coded in the client). Note that the server will not close anything and will continue running
(you can press CTRL-Break to stop the process)



Please note that you can open as many clients as you want and send messages to the server …



Please note also that the server accepts messages from all the clients as the notion of connection does not exist, each datagram/packet is an individual message by itself and all the datagrams are independent from each other.

Please also note that first client has etablished a connection to « localhost » which is translated to IPv4 loopback address 127.0.0.1, while the second client established a connection to 192.168.1.6 which is the present IPv4 address of my laptop. Of course you should check your IP with ipconfig and type the correct one otherwise the connection will be denied, and as there is no error processing routine in those simple Python codes, the client program will probably crash if there is no server to answer the request.

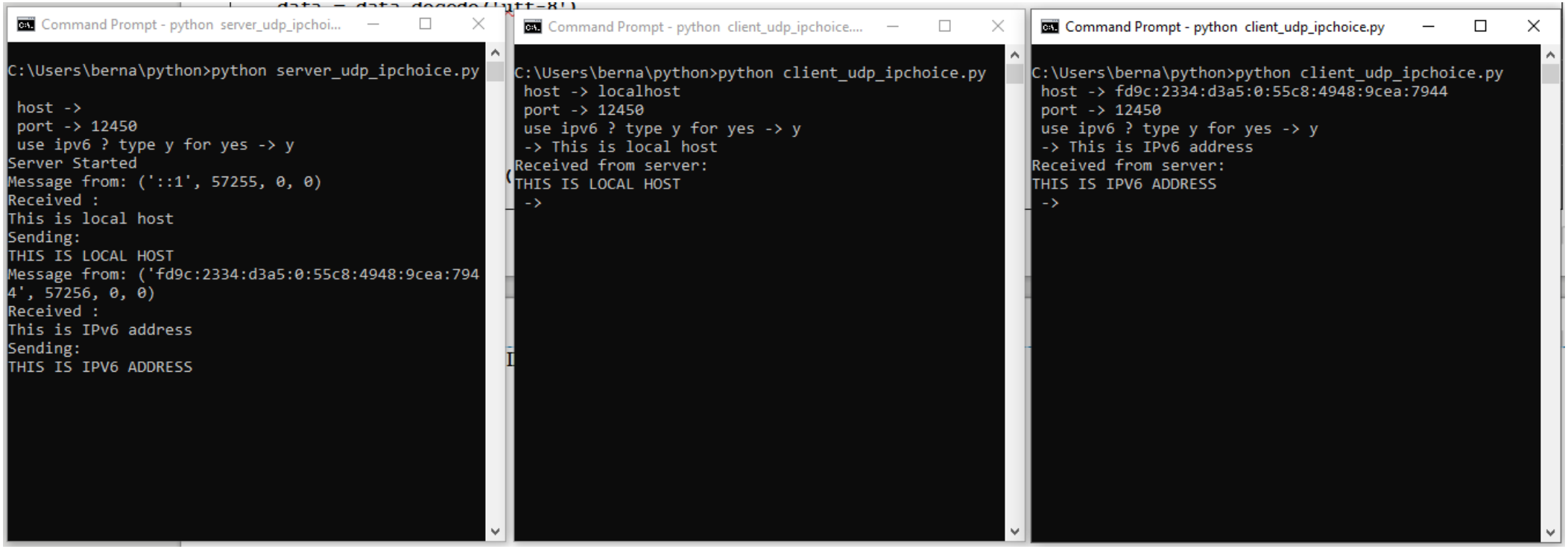
If I specified localhost on the server host prompt, then the client connecting to localhost would be accepted while the client connecting to 192.168.1.6 would be denied.

We have seen that IPv4 is by default. We can mofify a little bit the codes to implement a choice between the two IP versions :

```
# place this code in file client_udp_ipchoice.py :
import socket
host = input(" host -> ")
port = int(input(" port -> "))   # socket server port number
ipversion = input(" use ipv6 ? type y for yes -> ").lower()
ipv = socket.AF_INET
if ipversion=='y': ipv=socket.AF_INET6
server = (host,port)
client_socket = socket.socket(family=ipv,type=socket.SOCK_DGRAM)
message = input(" -> ")  # take input
while message.lower().strip() != 'bye':
    client_socket.sendto(message.encode(),server)
    data,addr = client_socket.recvfrom(1024)
    data=data.decode()  # receive response
    print('Received from server: ')
    print(data)  # show in terminal
    message = input(" -> ")  # again take input
client_socket.close()  # close the connection
```

```
# place this code in file server_udp_ipchoice.py :
import socket
host = input(" host -> ")
port = int(input(" port -> "))
ipversion = input(" use ipv6 ? type y for yes -> ").lower()
ipv = socket.AF_INET
if ipversion=='y': ipv=socket.AF_INET6
s = socket.socket(family=ipv,type=socket.SOCK_DGRAM)
s.bind((host, port))
print("Server Started")
while True:
    data, address = s.recvfrom(1024)
    data = data.decode('utf-8')
    if not data: break
    print("Message from: " + str(address))
    print("Received : ")
    print(data)
    data = data.upper()
    print("Sending: ")
    print(data)
    s.sendto(data.encode('utf-8'),address)
s.close()
```

So now we can try the same with IPv6 …



Once again we see that the server accepts datagram either from localhost (translated to ::1) as from the real IPv6 address.
Of course client and server must use the same Ipversion otherwise connection will be denied by server and client program will crash.

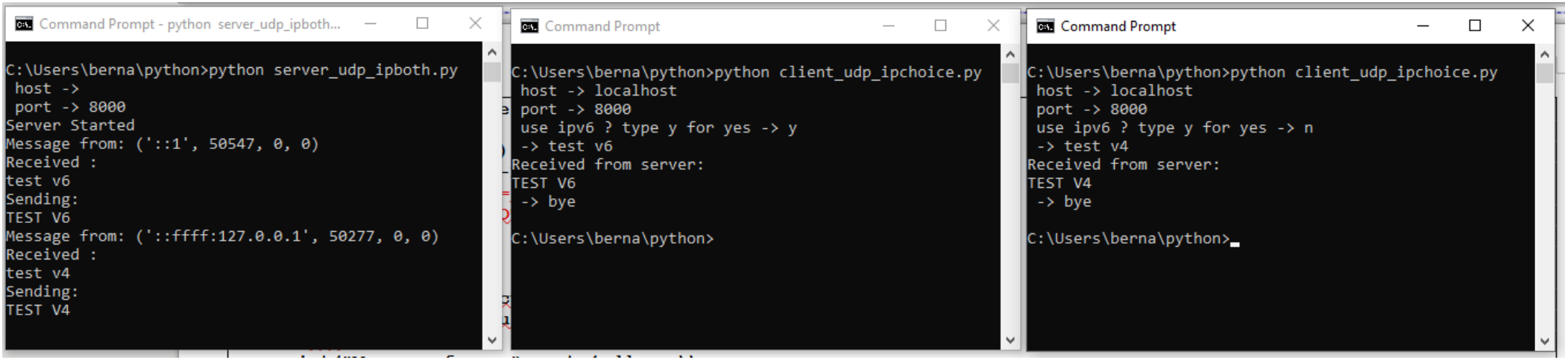
But there should be a solution to implement a server that accepts both IPv4 and IPv6 clients ? No ?
Solution found in forums …

https://stackoverflow.com/questions/65389606/how-to-handle-both-ipv4-and-ipv6-client-request

The solution is to set correct socket options (using the setsockopt() command)

```
# place this code in file server_udp_ipboth.py :
import socket
host = input(" host -> ")
port = int(input(" port -> "))
s = socket.socket(family=socket.AF_INET6,type=socket.SOCK_DGRAM)
s.setsockopt(socket.IPPROTO_IPV6,socket.IPV6_V6ONLY,0)
s.bind((host, port))
print("Server Started")
while True:
    data, address = s.recvfrom(1024)
    data = data.decode('utf-8')
    if not data: break
    print("Message from: " + str(address))
    print("Received : ")
    print(data)
    data = data.upper()
    print("Sending: ")
    print(data)
    s.sendto(data.encode('utf-8'),address)
s.close()
```

Of course this is not yet perfect, we normally have to check before implementing this if API supports dual stack protocol …
In fact, what happen is that API will translate IPv4 address into an IPv6 to which the server will respond … Here it works !!



Now we will see the clear difference between UDP which is connectionless and TCP which establish a connection between client and server using the 3-way handshake protocol.

Socket programming with TCP

Client must contact server

- server process must first be running
- server must have created socket (door) that welcomes client's contact

Client contacts server by:

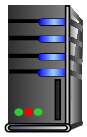
- Creating TCP socket, specifying IP address, port number of server process
- *when client creates socket:* client TCP establishes connection to server TCP

- when contacted by client, *server TCP creates new socket* for server process to communicate with that particular client
  - allows server to talk with multiple clients
  - *source* port numbers used to distinguish clients (more in Chap 3)

Application viewpoint

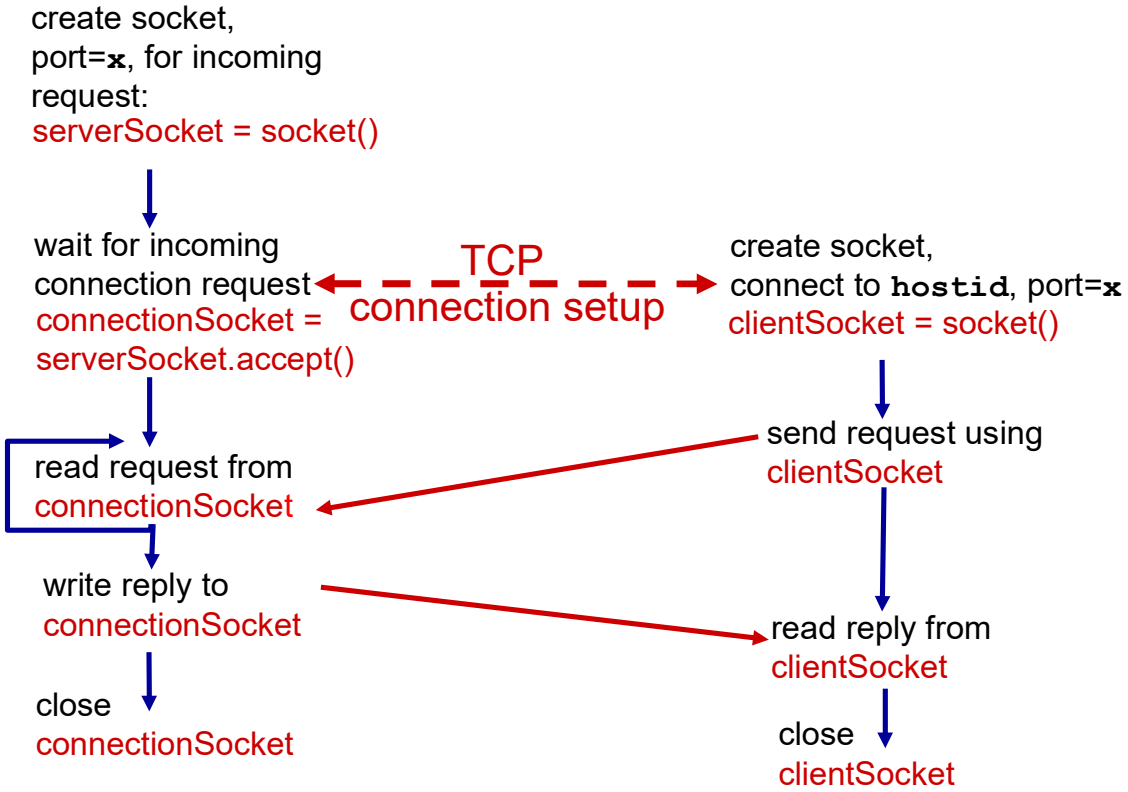
TCP provides reliable, in-order byte-stream transfer ("pipe") between client and server processes

Client/server socket interaction: TCP



server (running on hostid)

client 



Example app: TCP client

Python TCPClient

```
from socket import *
serverName = 'servername'
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName,serverPort))
sentence = raw_input('Input lowercase sentence:')
clientSocket.send(sentence.encode())
modifiedSentence = clientSocket.recv(1024)
print ('From Server:', modifiedSentence.decode())
clientSocket.close()
```

create TCP socket for server, remote port 12000

No need to attach server name, port

Example app: TCP server

Python TCPServer

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind(('',serverPort))
serverSocket.listen(1)
print 'The server is ready to receive'
while True:
    connectionSocket, addr = serverSocket.accept()

    sentence = connectionSocket.recv(1024).decode()
    capitalizedSentence = sentence.upper()
    connectionSocket.send(capitalizedSentence.
                                        encode())
    connectionSocket.close()
```

create TCP welcoming socket →

server begins listening for incoming TCP requests →

loop forever →

server waits on accept() for incoming requests, new socket created on return →

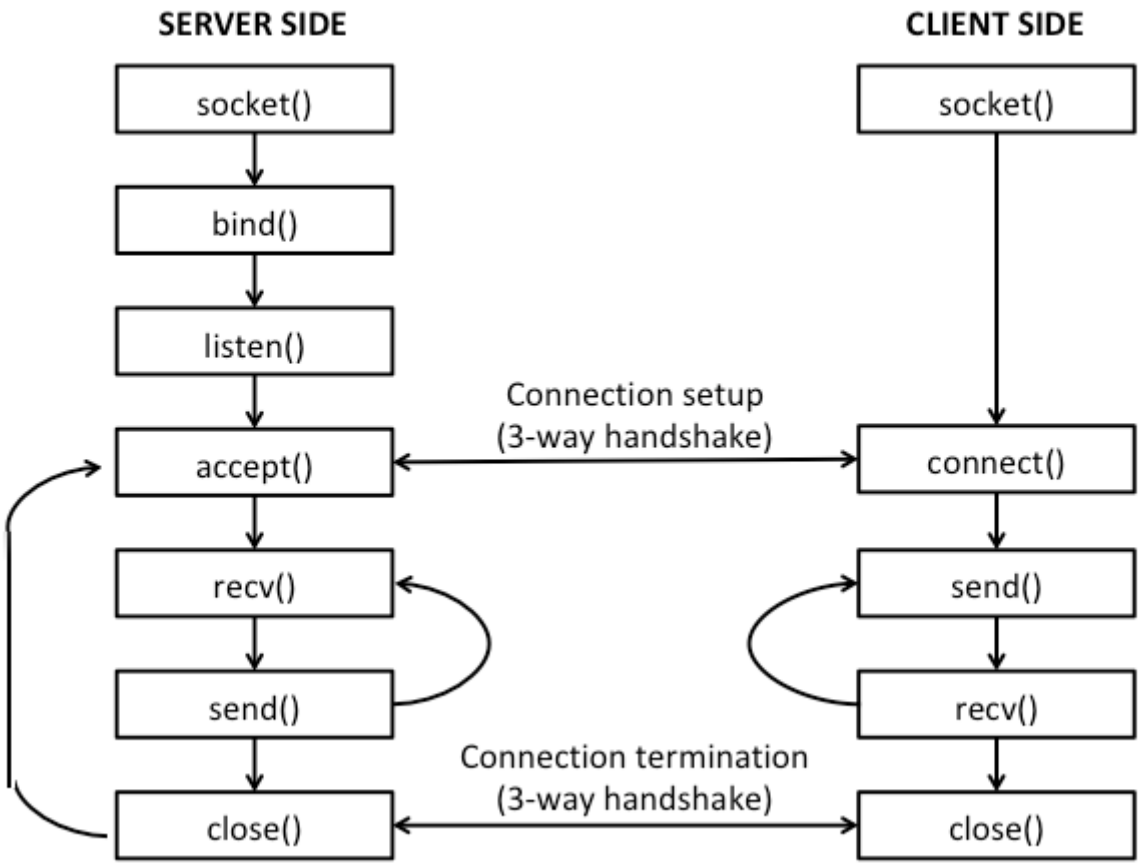
read bytes from socket (but not address as in UDP) →

close connection to this client (but *not* welcoming socket) →

```
# place this code in file client_tcp_simple.py :
import socket
host = input(" host -> ")
port = int(input(" port -> "))   # socket server port number
server = (host,port)
client_socket = socket.socket(family=socket.AF_INET,type=socket.SOCK_STREAM)
client_socket.connect(server)  # connect to the server
message = input(" -> ")  # take input
while message.lower().strip() != 'bye':
    client_socket.send(message.encode())  # send message
    data,addr = client_socket.recvfrom(1024)
    data=data.decode()  # receive response
    print('Received from server: ')
    print(data)  # show in terminal
    message = input(" -> ")  # again take input
client_socket.close()  # close the connection
```

```
# place this code in file server_tcp_simple.py :
import socket
host = input(" host -> ")
port = int(input(" port -> "))
s = socket.socket()
s.bind((host, port))
print("Server Started")
s.listen(1)
while True:
    conn, address = s.accept()  # accept new connection
    print("Connection from: " + str(address))
    while True:
        data = conn.recv(1024).decode('utf-8')
        if not data: break
        print("Message from: " + str(address))
        print("Received : ")
        print(data)
        data = data.upper()
        print("Sending: ")
        print(data)
        conn.send(data.encode('utf-8'))
    conn.close()
    print("Connection closed")
s.close()
```

This is the schematic of TCP :



Let’s first compare the two client codes : UDP and TCP :

```
# place this code in file client_udp_simple.py :
import socket
host = input(" host -> ")
port = int(input(" port -> "))   # socket server port number
server = (host,port)
client_socket = socket.socket(family=socket.AF_INET,type=socket.SOCK_DGRAM)
message = input(" -> ")  # take input
while message.lower().strip() != 'bye':
    client_socket.sendto(message.encode(),server)
    data,addr = client_socket.recvfrom(1024)
    data=data.decode()  # receive response
    print('Received from server: ')
    print(data)  # show in terminal
    message = input(" -> ")  # again take input
client_socket.close()  # close the connection
```

```
# place this code in file client_tcp_simple.py :
import socket
host = input(" host -> ")
port = int(input(" port -> "))   # socket server port number
server = (host,port)
client_socket = socket.socket(family=socket.AF_INET,type=socket.SOCK_STREAM)
client_socket.connect(server)  # connect to the server
message = input(" -> ")  # take input
while message.lower().strip() != 'bye':
    client_socket.send(message.encode())  # send message
    data,addr = client_socket.recvfrom(1024)
    data=data.decode()  # receive response
    print('Received from server: ')
    print(data)  # show in terminal
    message = input(" -> ")  # again take input
client_socket.close()  # close the connection
```

Let’s first compare the two client codes : UDP and TCP :
There is just one more line in TCP : the connect(server) command.

This initiates the TCP handshake.
The programmer doesn’t have to take care about the 3-way handshake, because the Windows API does everything …

Also note that the sento() comand doesn’t specify the server anymore as it has been associated with the socket through the connect() command.

The server code is however completely different …

```
# place this code in file server_tcp_simple.py :
import socket
host = input(" host -> ")
port = int(input(" port -> "))
s = socket.socket()
s.bind((host, port))
print("Server Started")
s.listen(1)
while True:
    conn, address = s.accept()  # accept new connection
    print("Connection from: " + str(address))
    while True:
        data = conn.recv(1024).decode('utf-8')
        if not data: break
        print("Message from: " + str(address))
        print("Received : ")
        print(data)
        data = data.upper()
        print("Sending: ")
        print(data)
        conn.send(data.encode('utf-8'))
    conn.close()
    print("Connection closed")
s.close()
```

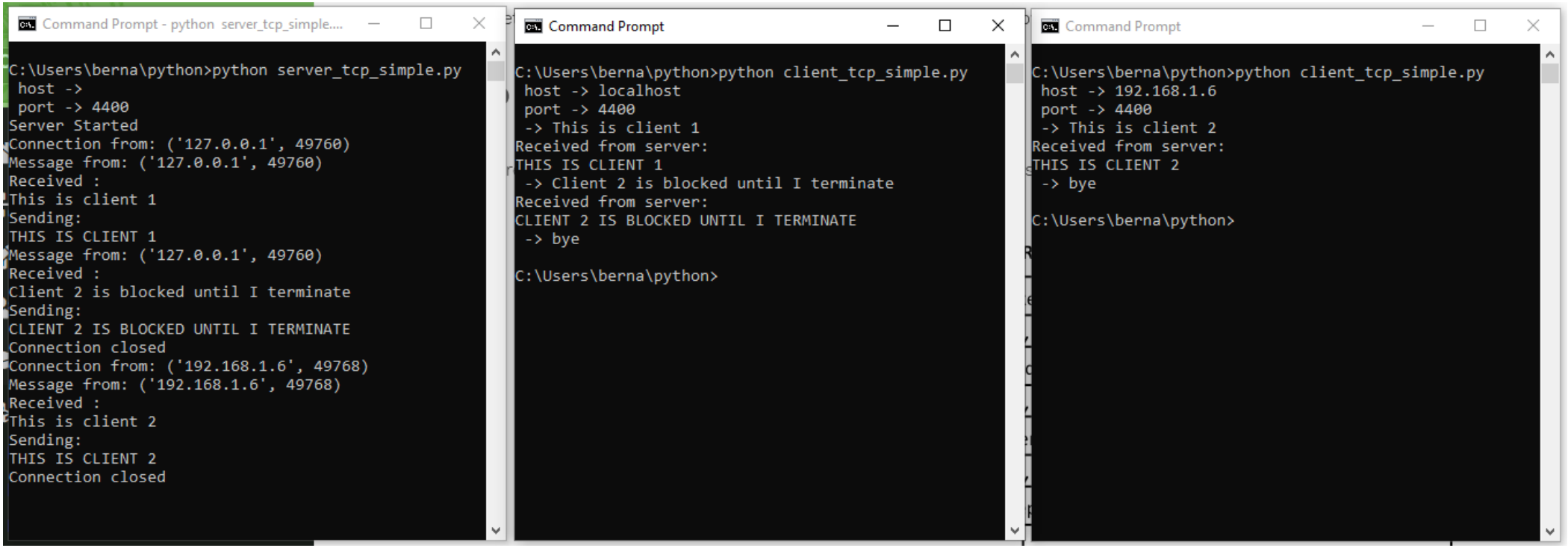
The bind() commend that makes association between socket and its port number is immediately followed by a listen() command.
The parameter of listen() is arbitrary and is not necessary.

The accept() command is the counterpart to the connect() from the client. It establishes the connexion. But please note that connextion is requested and initiated by the client, the server is just waiting for a connection.

The accept() creates a new structure called a connection, that will replace the server for the receiving and sending commands.

The code contains two imbricated loops. Server will continue dialog with client until client close the connection (which is the not data test).
Then server will come back to the outer loop waiting for a new connection.

Please note that as this code has no multithreading, if a second client try to connect while server is busy with first client, then this second client will be blocked until first client stops its connection.

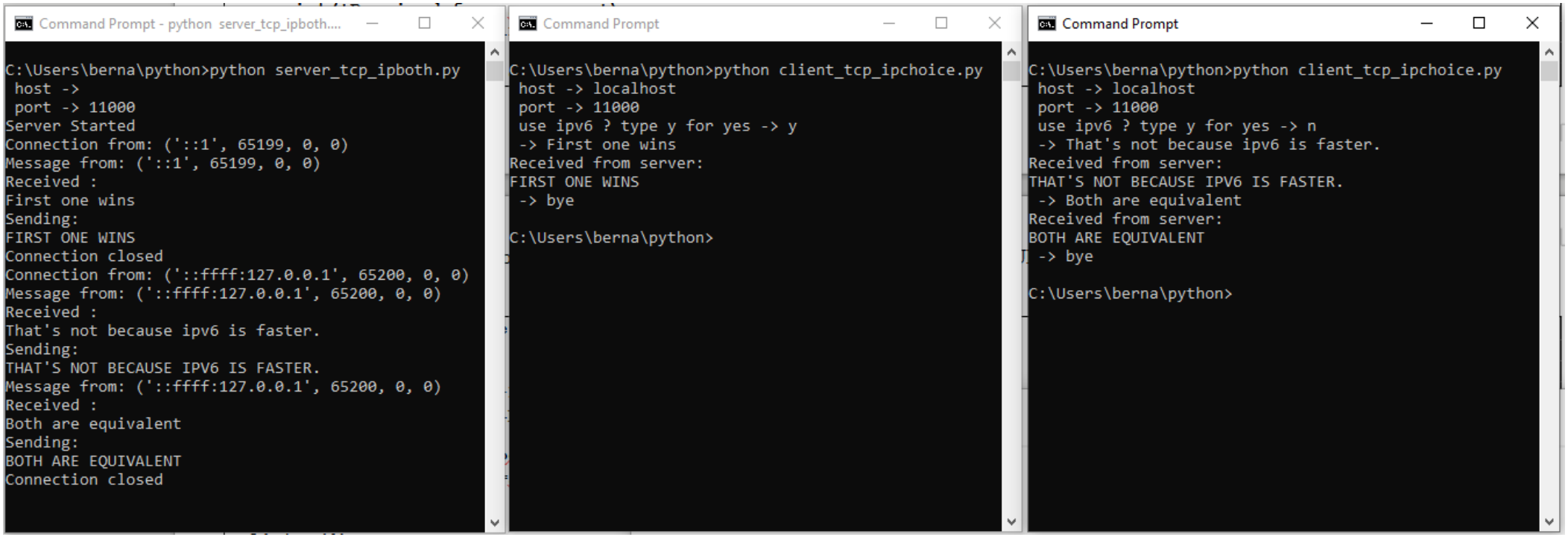


Of course these codes can be adapted to allow the choice between IPv4 and IPv6 like in UDP

```
# place this code in file client_tcp_ipchoice.py :
import socket
host = input(" host -> ")
ipversion = input(" use ipv6 ? type y for yes -> ").lower()
ipv = socket.AF_INET
if ipversion=='y': ipv=socket.AF_INET6
server = (host,port)
client_socket = socket.socket(family=ipv,type=socket.SOCK_STREAM)
client_socket.connect(server)  # connect to the server
message = input(" -> ")  # take input
while message.lower().strip() != 'bye':
    client_socket.send(message.encode())  # send message
    data,addr = client_socket.recvfrom(1024)
    data=data.decode()  # receive response
    print('Received from server: ')
    print(data)  # show in terminal
    message = input(" -> ")  # again take input
client_socket.close()  # close the connection
```

And we’ll go directly to the solution where the server accepts both IPv4 and IPv6 clients like in UDP.

```
# place this code in file server_tcp_ipboth.py :
import socket
host = input(" host -> ")
port = int(input(" port -> "))
s = socket.socket(family=socket.AF_INET6)
s.setsockopt(socket.IPPROTO_IPV6,socket.IPV6_V6ONLY,0)
s.bind((host, port))
s.listen(1)
while True:
    conn, address = s.accept()  # accept new connection
    print("Connection from: " + str(address))
    while True:
        data = conn.recv(1024).decode('utf-8')
        if not data: break
        print("Message from: " + str(address))
        print("Received : ")
        print(data)
        data = data.upper()
        print("Sending: ")
        print(data)
        conn.send(data.encode('utf-8'))
    conn.close()
    print("Connection closed")
s.close()
```

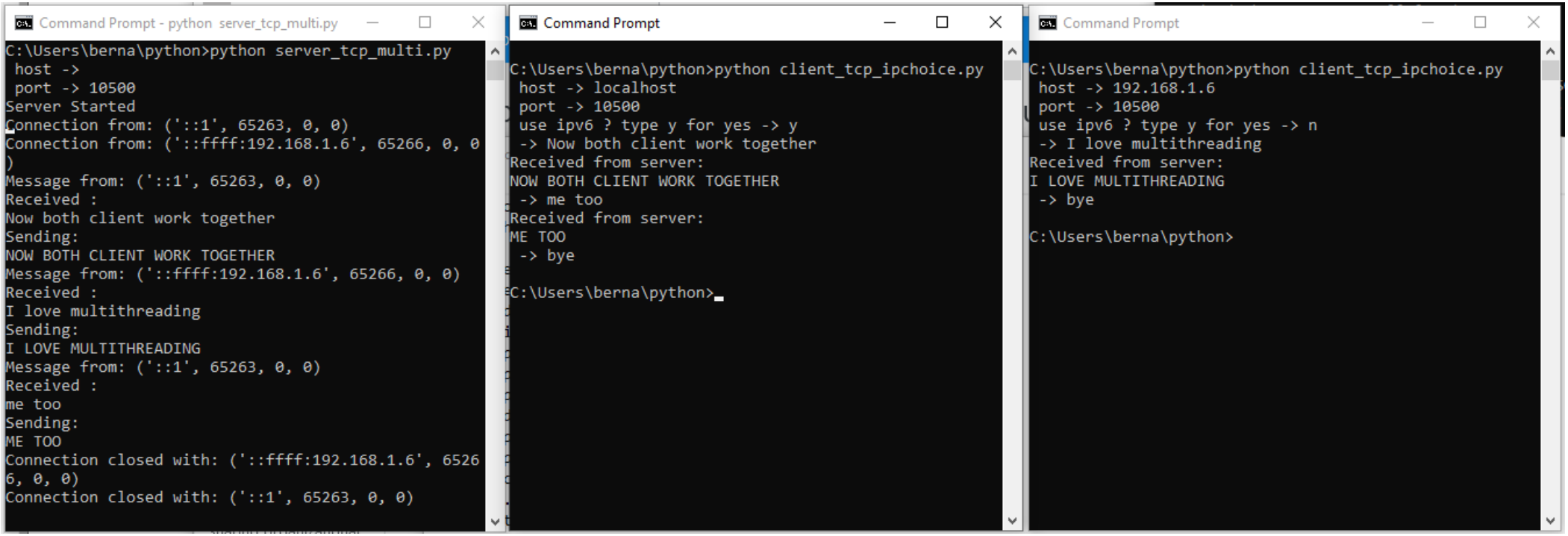


Once again it seems impossible to accept and respond to more than one client at the same time.
And once again the solution is quickly found in forums …

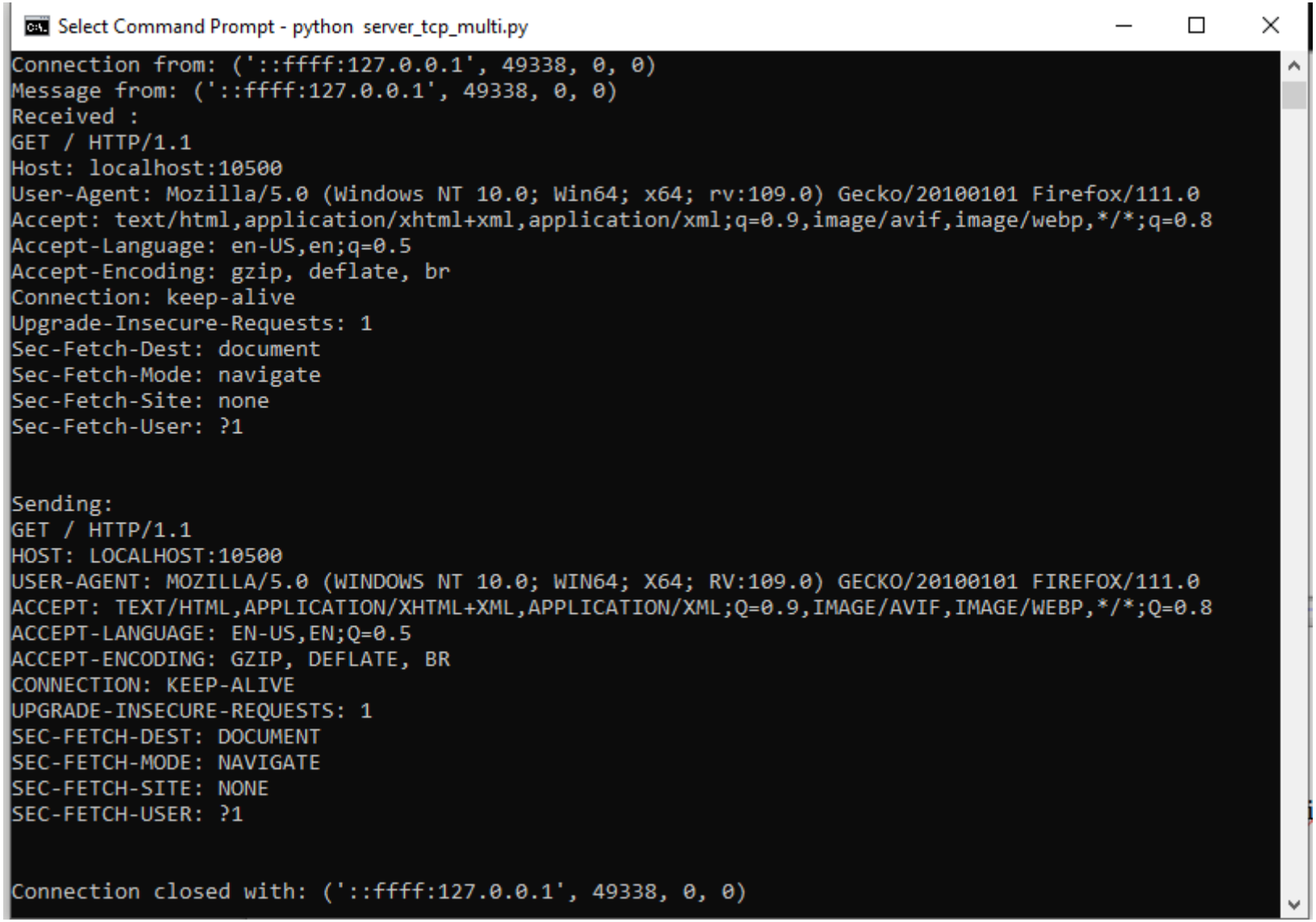
Fortunately … Python allows multithread programming, so the following one is a much better server …
Praise be Python … I don't know if it's so easy with another programming langage :)

```
# place this code in file server_tcp_multi.py :
import socket
import threading

def on_new_client(conn,addr):
    while True:
        data = conn.recv(1024).decode('utf-8')
        if not data: break
        print("Message from: " + str(address))
        print("Received : ")
        print(data)
        data = data.upper()
        print("Sending: ")
        print(data)
        conn.send(data.encode('utf-8'))
    conn.close()
    print("Connection closed")

host = input(" host -> ")
port = int(input(" port -> "))
s = socket.socket(family=socket.AF_INET6)
s.setsockopt(socket.IPPROTO_IPV6,socket.IPV6_V6ONLY,0)
s.bind((host, port))
print("Server Started")
s.listen(1)
while True:
    conn, address = s.accept()  # accept new connection
    print("Connection from: " + str(address))
    x = threading.Thread(target=on_new_client, args=(conn,address,))
    x.start()
s.close()
```

def is a procedure declaration
Each time the server receives a connection from a new client, it starts a new thread with the connection to that client …


Command Prompt - python server_tcp_multi.py
C:\Users\berna\python>python server_tcp_multi.py
host ->
port -> 10500
Server Started
Connection from: ('::1', 65263, 0, 0)
Connection from: ('::ffff:192.168.1.6', 65266, 0, 0)
Message from: ('::1', 65263, 0, 0)
Received :
Now both client work together
Sending:
NOW BOTH CLIENT WORK TOGETHER
Message from: ('::ffff:192.168.1.6', 65266, 0, 0)
Received :
I love multithreading
Sending:
I LOVE MULTITHREADING
Message from: ('::1', 65263, 0, 0)
Received :
me too
Sending:
ME TOO
Connection closed with: ('::ffff:192.168.1.6', 65266, 0, 0)
Connection closed with: ('::1', 65263, 0, 0)
Command Prompt
C:\Users\berna\python>python client_tcp_ipchoice.py
host -> localhost
port -> 10500
use ipv6 ? type y for yes -> y
-> Now both client work together
Received from server:
NOW BOTH CLIENT WORK TOGETHER
-> me too
Received from server:
ME TOO
-> bye
C:\Users\berna\python>
Command Prompt
C:\Users\berna\python>python client_tcp_ipchoice.py
host -> 192.168.1.6
port -> 10500
use ipv6 ? type y for yes -> n
-> I love multithreading
Received from server:
I LOVE MULTITHREADING
-> bye
C:\Users\berna\python>

Amazingly, by typing
localhost:10500
in the address bar of a browser …
Of course our simple server received the request … but as it is not an HTTP server, it didn't send an adequate answer (just echoing the brower request in upper case), so nothing happens on the browser window …



And now the final test … run the server on one computer and the client on an other one.

Be careful of a few things …
- Your computer firewall will probably block the connection, so you can :
  - Either temporarily disable your firewall (not recommanded if you are connected on internet)
  - Or set the connection to PRIVATE (it's in the connection settings), that means that your firewall will assume that any connection within your LAN is secure

- Most routers, including ISP boxes are capable of IPv6 routing but not by default within hosts on your LAN. Moreover they don't have a DNS translation in IPv6. Only IPv4 translation. You can check that by typing
nslookup <host name of your computer>
and hostname of your computer can be get with
whoami

So if you want to use IPv6, you have to type the complete IPv6 address it received from the LAN DHCP server.
A good solution is always to ping the other computer before trying … if ping works then python sockets will work too…

Here are a few things I tested …
Firrst this is ipconfig of both machines :

ipconfig from client

```
Ethernet adapter Ethernet:

   Connection-specific DNS Suffix  . : dlink
   IPv6 Address. . . . . . . . . . . : fd4d:3aac:67a0:0:c5b0:442e:e4c0:23c2
   Temporary IPv6 Address. . . . . . : fd4d:3aac:67a0:0:1ff:fbf9:e5a6:3872
   Link-local IPv6 Address . . . . . : fe80::2804:30a6:34b9:6459%13
   IPv4 Address. . . . . . . . . . . : 192.168.0.100
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . : 192.168.0.1

Wireless LAN adapter WiFi:

   Connection-specific DNS Suffix  . : lan
   IPv6 Address. . . . . . . . . . . : 2a02:a03f:c0d8:2800:3314:2513:a5fa:6f86
   IPv6 Address. . . . . . . . . . . : fd9c:2334:d3a5:0:55c8:4948:9cea:7944
   Temporary IPv6 Address. . . . . . : 2a02:a03f:c0d8:2800:dc1a:4a8e:cb61:426d
   Temporary IPv6 Address. . . . . . : fd9c:2334:d3a5:0:dc1a:4a8e:cb61:426d
   Link-local IPv6 Address . . . . . : fe80::a2ad:e0ad:c3c3:b48b%19
   IPv4 Address. . . . . . . . . . . : 192.168.1.6
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . : fe80::a6b1:e9ff:febd:5b12%19
                                       192.168.1.1
```

ipconfig from server

```
Carte Ethernet Ethernet :

   Suffixe DNS propre à la connexion. . . : dlink
   Adresse IPv6. . . . . . . . . . . . . .: fd4d:3aac:67a0:0:cc8a:105e:ae6d:afe0
   Adresse IPv6 temporaire . . . . . . . .: fd4d:3aac:67a0:0:6d2c:f202:c723:a7b4
   Adresse IPv6 de liaison locale. . . . .: fe80::cc8a:105e:ae6d:afe0%15
   Adresse IPv4. . . . . . . . . . . . . .: 192.168.0.101
   Masque de sous-réseau. . . . . . . . . : 255.255.255.0
   Passerelle par défaut. . . . . . . . . : 192.168.0.1

Carte réseau sans fil Wi-Fi :

   Suffixe DNS propre à la connexion. . . : lan
   Adresse IPv6. . . . . . . . . . . . . .: 2a02:a03f:c0d8:2800:fc99:ed4:7644:2522
   Adresse IPv6. . . . . . . . . . . . . .: fd9c:2334:d3a5:0:fc99:ed4:7644:2522
   Adresse IPv6 temporaire . . . . . . . .: 2a02:a03f:c0d8:2800:48a9:6174:f57a:e610
   Adresse IPv6 temporaire . . . . . . . .: fd9c:2334:d3a5:0:48a9:6174:f57a:e610
   Adresse IPv6 de liaison locale. . . . .: fe80::fc99:ed4:7644:2522%4
   Adresse IPv4. . . . . . . . . . . . . .: 192.168.1.13
   Masque de sous-réseau. . . . . . . . . : 255.255.255.0
   Passerelle par défaut. . . . . . . . . : fe80::a6b1:e9ff:febd:5b12%4
                                            192.168.1.1
```

Both machines are connected to 2 different routers.
- One through Ethernet cable, it's a Dlink DIR-910 router with no WAN connection to internet during test. This is an old router (bios dating 2013)
- One through WiFi, its a bbox3 from ISP

First note that both interfaces provide different IP addresses.
Both routers provides DHCP in v4 and in v6.
Only bbox provides additional information like DNS, but we will see that bbox is unable to route LAN in v6 while old Dlink can.

Test sessions on client

```
C:\Users\berna\python>python client_tcp_ipchoice.py
 host -> LAPTOP-K9MVJBE2
 port -> 3000
 use ipv6 ? type y for yes -> n
 -> Test
Received from server:
TEST
 -> bye

C:\Users\berna\python>python client_tcp_ipchoice.py
 host -> 192.168.0.101
 port -> 3000
 use ipv6 ? type y for yes -> n
 -> Test2
Received from server:
TEST2
 -> bye

C:\Users\berna\python>python client_tcp_ipchoice.py
 host -> fd4d:3aac:67a0:0:cc8a:105e:ae6d:afe0
 port -> 3000
 use ipv6 ? type y for yes -> y
 -> test3
Received from server:
TEST3
 -> bye

C:\Users\berna\python>python client_tcp_ipchoice.py
 host -> 2a02:a03f:c0d8:2800:fc99:ed4:7644:2522
 port -> 3000
 use ipv6 ? type y for yes -> y
Traceback (most recent call last):
  File "C:\Users\berna\python\client_tcp_ipchoice.py", line 9, in <module
>
    client_socket.connect(server)  # connect to the server
TimeoutError: [WinError 10060] A connection attempt failed because the co
nnected party did not properly respond after a period of time, or establi
shed connection failed because connected host has failed to respond

C:\Users\berna\python>
```

Transcript on server

```
D:\Python>python server_tcp_multi.py
 host ->
 port -> 3000
Server Started
Connection from: ('::ffff:192.168.1.6', 61234, 0, 0)
Message from: ('::ffff:192.168.1.6', 61234, 0, 0)
Received :
Test
Sending:
TEST
Connection closed with: ('::ffff:192.168.1.6', 61234, 0, 0)
Connection from: ('::ffff:192.168.0.100', 61315, 0, 0)
Message from: ('::ffff:192.168.0.100', 61315, 0, 0)
Received :
Test2
Sending:
TEST2
Connection closed with: ('::ffff:192.168.0.100', 61315, 0, 0)
Connection from: ('fd4d:3aac:67a0:0:1ff:fbf9:e5a6:3872', 61373, 0, 0)
Message from: ('fd4d:3aac:67a0:0:1ff:fbf9:e5a6:3872', 61373, 0, 0)
Received :
test3
Sending:
TEST3
Connection closed with: ('fd4d:3aac:67a0:0:1ff:fbf9:e5a6:3872', 61373, 0, 0)
```

1. Test in IPv4 while providing server hostname. We see that bbox makes DNS translation, on server received address is 192.168.1.13
2. Test in IPv4 with hard IP address on Dlink server.
3. Test in IPv6 with hard IPv6 address on Dlink server
4. Test in IPv6 with hard IPv6 address on bbox server. This one fails and crashes the client.

Of course all of you are encouraged to test with your own hardware …

And …

=> Mandatory homework : report on this lab

Test a client/server system in Python
You can replace the transformation to uppercase by the server with something else. A good exercise should be for example to send a small text file or simulate a small 2-players game.
You can go further that was presented here
You can compare the possibilities with your own hardware and/or OS, as everything here is dependent on that
There is no obligation of result. If you fail, then just explain what you think is the problem.

This shoud be real report (not only source files)

Deadline : before course session 7 (2 weeks)

2024 :
Write a network application : for example a multiplayer game
- You can use the client/server paradigm (which is probably easier) or peer-to-peer
- You can use TCP or UDP sockets (which is probably easier) or any other access to lower layer protocols
- Programming language should be python
- Your application must work at least on distributed machines within the same LAN
- Deliverables are a full pdf report plus the source code
- Work can be done individually or by groups of 2 or 3 maximum (the more in the group the higher are expectations
- Deadline is Sunday before the break at the end of semester (in 2024 : June 2)