

Exercise 1.1

Objectives:

- Low-level network programming with sockets
- How to connect to a TCP server

Sockets are programming abstraction for network code. These are the endpoints for a communication channel between two devices. To create a socket in python:

```
import socket s = socket.socket(addr_family, type)
```

addr_family can be:

```
socket.AF_INET # IPv4 socket.AF_INET6 # IPv6
```

while type can be:

```
socket.SOCK_STREAM # TCP socket.SOCK_DGRAM # UDP
```

An example of a TCP client requesting, saving and printing data:

```
from socket import *
s = socket(AF_INET, SOCK_STREAM) # create socket
s.connect(("www.python.org", 80)) # connect to network, host tuple
s.send("GET /index.html HTTP/1.0") # send request

chunks = [] # create an array to put data into
while True:
    chunk = s.recv(16384) # receive 16384 bytes each iteration
    if not chunk: break # if null is reached, break
    chunks.append(chunk) # append each chunk to array
s.close() # close connection after null is reached
response = "".join(chunks) # join individual chunks to create string
print response # look at the output
```

In this exercise, we briefly look at how to establish connections to a TCP server.

(a) Connecting to a TCP Server

The HTTP protocol used by the web is based on TCP. Experiment with the protocol by manually connecting to a web server, issuing a request, and reading the response.

```
>>> from socket import *
>>> s = socket(AF_INET, SOCK_STREAM)
>>> s.connect(("www.python.org", 80))
>>> s.send("GET /index.html HTTP/1.0\r\n\r\n")
>>> chunks = []
>>> while True:
```

```

        chunk = s.recv(16384)
        if not chunk: break
        chunks.append(chunk)
>>> s.close()
>>> response = "".join(chunks)
>>> print response
... look at the output ...
>>>

```

This code is fairly typical for TCP client code. Once connected to a server, use `send()` to send request data. To read a response, you will typically have to read data in chunks with multiple `recv()` operations. `recv()` returns an empty string to signal the end of data (i.e., if the server closed its end of the connection). Recall that using the string `join()` method is significantly faster than using string concatenation (+) to join string fragments together.

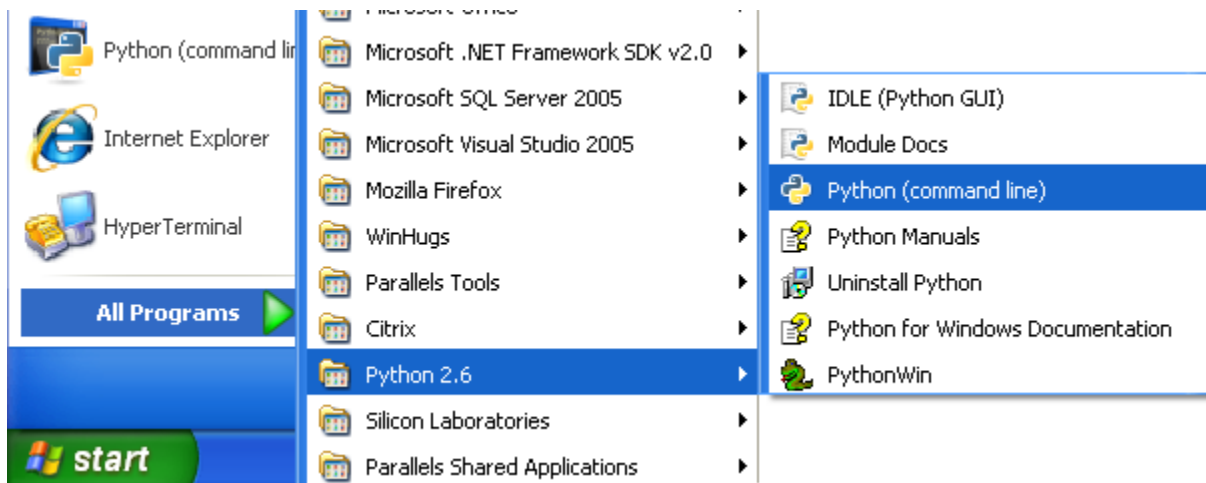
Exercise 1.2

Objectives:

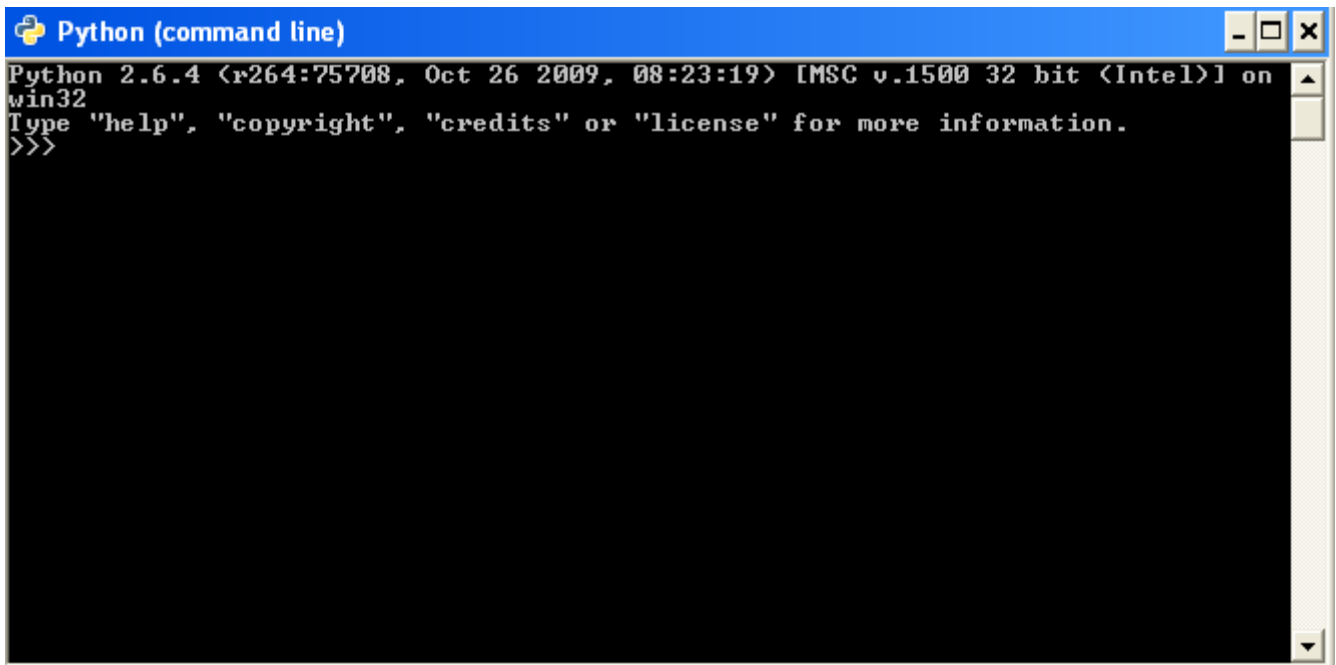
- Experiment with TCP client and server connections
- Learn about the web

In this exercise, we experiment with setting up a network connection between two different Python interpreters.

To do this exercise, you need to start a different Python interpreter than the one you are using with IDLE. On Windows, go to the "Start" menu and run the "Python (command line)" program. For example:



This should pop up a command window where you can directly interact with a Python interpreter.



On Linux and Macintosh, open a new terminal window (e.g., xterm) and run Python at the command line by typing 'python'.

(a) Waiting for a Connection

In the new Python interpreter that you just opened, type the following commands to create a very simple network server:

```
>>> from socket import *
>>> s = socket(AF_INET, SOCK_STREAM)
>>> s.bind(("", 15000))
>>> s.listen(5)
>>> c,a = s.accept() # c is new socket created for this connection while a is the
network/port addr of client as a tuple
```

These statements create a server that is now waiting for an incoming connection. The `accept()` operation blocks forever until a connection arrives---so, you won't be able to do anything else until that happens.

(b) Making a Connection

Using IDLE, type the following statements to make a connection to your server program. Keep in mind, these statements are being typed into a *different* Python interpreter than the one that's running in part (a).

```
>>> from socket import *
>>> s = socket(AF_INET, SOCK_STREAM)
>>> s.connect(("localhost", 15000))
>>>
```

Immediately after you issue the `connect()` operation, your server program in part (a) should suddenly return back to the Python `>>>` prompt. Type the following statements in the server to see the result.

```
>>> c
<socket._socketobject object at 0x10028d910>
>>> a
('127.0.0.1', 62793)
>>>
```

The variable `c` returned by `accept()` is a socket object that's connected to the remote client. The variable `a` is a tuple containing the IP address and port number of the remote client.

(c) Sending Some Data

In IDLE (the client), type the following statement to send a "Hello World" message to the server:

```
>>> s.send("Hello World")
11
>>>
```

In the server, type the following to receive the message:

```
>>> data = c.recv(1024)
>>> data
'Hello World'
>>>
```

Have the server send a message back to the client:

```
>>> c.send("Hello Yourself")
14
>>>
```

Have the client receive the server's response. In IDLE, type

```
>>> resp = s.recv(1024)
>>> resp
'Hello Yourself'
>>>
```

Observe that you are sending data back and forth between the two Python interpreters. Try more `send()` and `recv()` operations if you want to send more data.

(d) Closing the Connection

In IDLE (the client), issue a `recv()` operation to wait for more data. This should block since the server hasn't actually sent any additional data yet.

```
>>> s.recv(1024)
```

In the server, send a goodbye message and close the socket.

```
>>> c.send("Goodbye")
7
>>> c.close()
>>>
```

When the `send()` operation completes, the client should display the 'Goodbye' message. For example:

```
>>> s.recv(1024)    (from before)
'Goodbye'
>>>
```

Try reading more data and see what happens:

```
>>> s.recv(1024)
''
>>>
```

You'll notice that the `recv()` operation returns with an empty string. This is the indication that the server has closed the connection and that there is no more data. There is nothing more than the client can do with the socket at this point.

(e) Listening for a new connection

After the server has closed the client connection, it typically goes back to listening by issuing a new `accept()` command. Type this in the server:

```
>>> c,a = s.accept() # goes back to listening for more clients
```

Once again, the server now waits until a new connection arrives.

(f) Connecting with a Browser

Try connecting to your server program using your internet browser. Simply click on this link (or type it into the navigation bar):

<http://localhost:15000/index.html>

In your server program, you should see that the `accept()` operation has returned. Try reading the browser request and print it out.

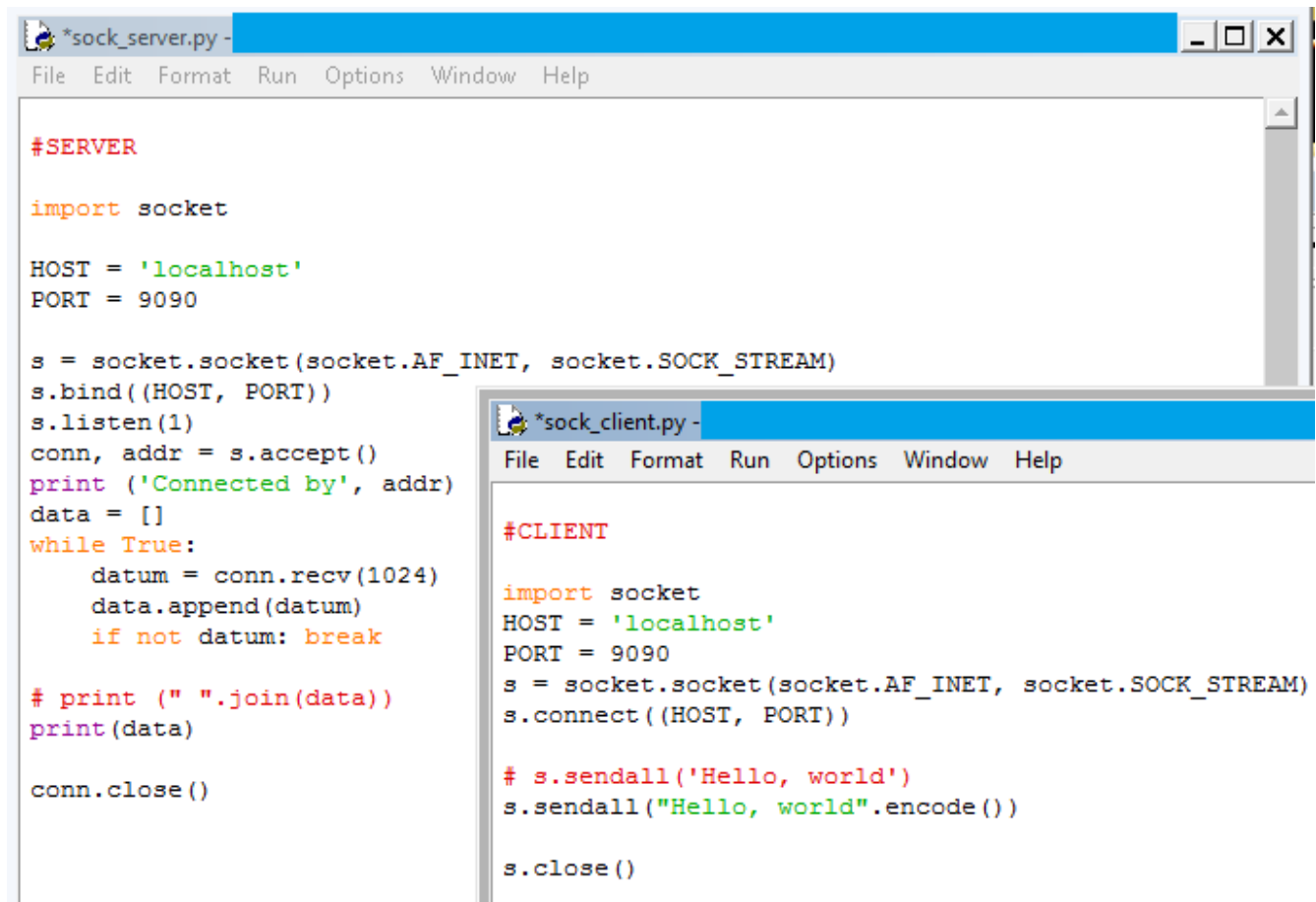
```
>>> request = c.recv(8192)
>>> print request
... look at the result ...
>>>
```

Send an HTML greeting back to the browser:

```
>>> c.send("HTTP/1.0 200 OK\r\n")
>>> c.send("Content-type: text/html\r\n")
>>> c.send("\r\n")
>>> c.send("<h1>Hello World</h1>")
>>> c.close()
>>>
```

Congratulations! You have now made an extremely lame web server.

Exercise 1.3



The image shows two overlapping Python IDE windows. The top window, titled '*sock_server.py', contains the following code:

```
#SERVER

import socket

HOST = 'localhost'
PORT = 9090

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((HOST, PORT))
s.listen(1)
conn, addr = s.accept()
print ('Connected by', addr)
data = []
while True:
    datum = conn.recv(1024)
    data.append(datum)
    if not datum: break

# print (" ".join(data))
print(data)

conn.close()
```

The bottom window, titled '*sock_client.py', contains the following code:

```
#CLIENT

import socket
HOST = 'localhost'
PORT = 9090
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((HOST, PORT))

# s.sendall('Hello, world')
s.sendall("Hello, world".encode())

s.close()
```