Appendix

This appendix is designed to give you a quick introduction to socket programming in Python. First things first, you should prepare a socket for your program to listen on. Indeed, if you don't have a socket, how will you be able to hear the client? If you have never worked with sockets before, this might sound a little daunting but don't fret. You will need to go through four steps to set up your first listening socket. You would be wise to not just blindly follow these instructions but to do your best to understand why we are doing this.

1. Actually create the socket. You can use the socket constructor for this (https://docs.python.org/2/library/socket.html#socket.socket). The default settings (to use `AF_INET` and `SOCK_STREAM`) should be fine.
2. This step is optional but recommended. By default, you can run into an annoying error with sockets already being in use when you repeatedly start your program for test purposes. You can use the `setsockopt` method to remedy this. You should set the `SO_REUSEADDR` to 1 to make sure the local address can be reused.
3. The socket has to be bound to a specific address and port. This is done using the `bind` method which takes a tuple consisting of the name of the host and the port. For now, you can just set the host to `localhost` and select your favourite number as the port. Usually, HTTP uses port 80, 8080 or 8081.
4. Lastly, you should set the socket to listen using the cleverly named `listen` method. This method normally takes an integer to determine how big the backlog should be, but you can set this to 0 for now.

To better understand how sockets work in Python, you can try the following experiment. Open two terminals and open a Python interpreter in one of them. Then create a socket using the instructions above. Remember the host and port you bind it to! Then, type the following into your Python interpreter:

```
>>> c, _ = socket_name.accept()
```

As you see, this blocks. That is to say, the execution of the program does not continue. Can you image why? If you thought "*gee, I guess that's because there's no incoming connections*", you would be right. Remember that second terminal window we opened? Remember the `nc` command from the first assignment? Let's use that to create an incoming connection for our lonely socket.

```
$ nc localhost <your_port>
```

At this point, you should see a prompt appear in your Python interpreter almost instantly. What this means is that it has received a connection and stored it in `c`. This is simply a new socket object. The `accept` method also returns the address on the other end, but we don't need that right now so we discard it by putting it in the `_` variable. For the uninitiated, that is the preferred way to treat data you do not want in Python.

You might be wondering what to do now. Well, the point of a socket is to be able to get data from one point to another over a network. So let's try that. The socket that was accepted and stored in the `c` variable features the `recv` method which is short for `receive`. Let's call it (with the argument that represents a number of bytes to receive) and see what happens.

```
>>> c.recv(512)
```

You should see it block again. This is because there is nothing in the socket which we can retrieve. To put something there, let's go back to the `nc` terminal we had open and type something. Anything will do. Just make sure it ends with a newline. You should see the Python interpreter unblock and spit out the string that you just entered into `nc`. Well done! You have created your first Python socket and sent data over it.

Now, what happens if we send data to the socket without having our Python script wait for data? That's not a problem. Try it for yourself, enter some data without having the `recv` method running. Then, when you're done, execute the `recv` method like we did before. What do you see?

Let's say that we would now like to send some data back to the user. This could be some text, an image, a complete HTTP response or a few seconds of ABBA's 1980 hit *'Happy new year'*. For this, you can simply use the `send` method of the socket. Remember that when we called the `accept` method on the listening socket, that created a new socket which links both sides. Try the following in your Python interpreter and see what happens in the `nc` instance:

```
>>> c.send("Hey there!")
```

Not bad, right? As you can see, it's quite easy to both send and receive data when you get the hang of it. If we just want to send data without listening for connections on a certain port, we can use largely the same process we used to set up out listening socket except for the `bind` and `listen` methods, which are replaced by a single `connect` method. This takes a tuple of host and port, just like `bind`. Apart from that, such a socket is functionally identical to what we have done here.