A network socket is an endpoint of an interprocess communication across a computer network. The Python Standard Library has a module called `socket` which provides a low-level internet networking interface. This interface is common across different programming languages since it uses OS-level system calls.

To create a socket, there is a function called `socket`. It accepts `family`, `type`, and `proto` arguments (see documentation for details). To create a TCP-socket, you should use `socket.AF_INET` or `socket.AF_INET6` for `family` and `socket.SOCK_STREAM` for `type.`
Here's a Python socket example:

```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

It returns a socket object which has the following main methods:

`bind()`

`listen()`

send()

recv()

`bind()`, `listen()` and `accept()` are specific for server sockets. `connect()` is specific for client sockets. `send()` and `recv()` are common for both types. Here is an example of Echo server from documentation:

```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind(('localhost', 50000))
s.listen(1)
conn, addr = s.accept()
while 1:
    data = conn.recv(1024)
    if not data:
        break
    conn.sendall(data)
conn.close()
```

Here we create a server socket, bind it to a localhost and 50000 port, and start listening for incoming connections. To accept an incoming connection we call `accept()` method which will block until a new client connects. When this happens, it creates a new socket and returns it together with the client's address. Then, in an infinite

```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(('localhost', 50000))
s.sendall('Hello, world')
data = s.recv(1024)
s.close()
print 'Received', repr(data)
```

Here instead of `bind()` and `listen()` it calls only `connect()` and immediately sends data to the server. Then it receives 1024 bytes back, closes the socket, and prints the received data.

All socket methods are blocking. For example, when it reads from a socket or writes to it the program can't do anything else. One possible solution is to delegate working with clients to separate threads. However, creating threads and switching contexts between them is not really a cheap operation. To address this problem, there is a so-called asynchronous way of working with

They are all about the same so let's create a server using Python select. Here's a Python `select` example:

```
import select, socket, sys, Queue
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.setblocking(0)
server.bind(('localhost', 50000))
server.listen(5)
inputs = [server]
outputs = []
message_queues = {}

while inputs:
    readable, writable, exceptional = select.select(
        inputs, outputs, inputs)
    for s in readable:
        if s is server:
            connection, client_address = s.accept()
```

```
            if s in outputs:
                outputs.remove(s)
            s.close()
            del message_queues[s]
```

As you can see, there is much more code than in the blocking Echo server. That is primarily because we have to maintain a set of queues for different lists of sockets, i.e. writing, reading, and a separate list for erroneous sockets.

`accept()`, adds a returned socket to `inputs` and adds a `Queue` for incoming messages which will be sent back. If there is another socket in inputs, then some messages have arrived and ready to be read so it reads them and places them into the corresponding queue.