Socket Programming Using Python

Python is the most versatile language and it has a vast range of library which used to almost every trending fields. As we know that, Python has easy syntax and user-friendly environment that makes development or data analysis straightforward.

Python is also compatible for the network programming and it consists of many libraries that offer higher-level access to specific application-level network protocols, such as HTTP, FTP, and so on. First we need to understand the basic terminologies of the networking.

What is Networking?

Computer Network is a term which focus to study the communication process among the more than one computer devices or system that are linked with each other. These linked devices exchange the information and share resources. So in order to exchange information, we must establish the network connection between the devices. For instance - Make a phone call must have network service provider.

here are four types of computer network.

- LAN (Local Area Network)
- WAN (Wide Area Network)
- MAN (Metropolitan Area Network)
- PAN (Personal Area Network)

Let's understand the brief introduction of above networks.

LAN (Local Area Network)

The local area network is used to connect the small number of systems located in a small geographical space such as college, small organization, offices, home, etc. It is more of a peer to peer connection. A LAN can be small in size, as well as huge, reaching from a home network with one user to an enterprise network with the many users and devices.

WAN (Wide Area Network)

The wide area network covers the more geographical area in contrast of local area network. It's architecture basically based on an assembly of many LAN's together to reach outside just peer to peer sharing. The internet is the best example of the wide area network.

The WAN uses the packet switching that is the method of the data transmission. In Packet Switching, message is broken into the several parts, called packets and that packets are transferred independently.

MAN (Metropolitan Area Network)

The Metropolitan Area Network is bigger than the area covered by the LAN but smaller than the WAN (wide area network). It interconnects the users with the computer resources in medium geographical range. **For example -** A city is connected using a single larger network.

PAN (Personal Area Network)

The Personal Area Network is used to transmit the data amongst the devices such as computer, telephones, tablets and personal digital assistants. PAN can be used for communication personal devices themselves or for connecting to a higher level network.

This is all about the different types of network. Now, we will understand the Network terminologies.

Computer Network Terminologies

These terminologies are very essential for the network programming. If you are not familiar with them then understand them for further tutorial.

Internet Protocol

The internet protocol (IP) is a set of rules, or addressing packets of data that they can travel across networks and responsible for dispatch the data packet to the correct destination.

For example - There are two friends **A** and **B**. Suppose **A** want to communicate to the **B**. Then, he wrote a letter and went to the **Post-Office** (A kind of a communication network). He putted the letter into envelop and submit it in the

Post Office for delivery to **B**. He attached the address of **B** so that the letter can be reached at the correct destination.

Now, **A** wants to send the script of 15 pages to **B**. So, there is a chance that one envelop cannot fit in a single envelop. So he decide to put each page in separate envelop. Now, there is chance the post office will not deliver the letter in the same order as they were supposed to be? Here the internet protocol plays important role.

There are two major protocols over the internet.

- UDP(User Datagram Protocol)
- TCP(Transmission Control Protocol)

User Datagram Protocol

The UDP sends the data over the internet as **datagrams**. It is a connectionless protocol. In the previous example, the 15 pages are referred as **15 datagrams**. Let's understand the following properties of UDP protocols.

- **Unreliable -** It is an unreliable protocol because it doesn't inform the sender the dataframe is sent successfully or not. When we send the datagram over the internet, there is no way to know if it is reached its destination or not.
- **Unordered -** We cannot predict the order of delivered packet. The receiver order can be differed from the sender dataframe.
- **Lightweight -** There is no connections, no ordering of messages.
- **Datagrams -** Datagrams are the data packet which sent individually and checked for integrity.

To overcome the above mentioned issue, there is another protocol called TCP.

TCP (Transmission Control Protocol)

The transfer control protocol is based on concept of **handshake**. It is the process where a host establishes the connection between interested hosts and therefore data transfer can be initiated.

For example - When we can call someone and say "Hello" and in replies other person replies "Hello". This is nothing but an assurance of connection has been established and now we can transfer the data over the network. This is the easiest example of handshake.

Let's understand the following properties of TCP.

- **Reliable -** It is more reliable than the UDP because it handles the message acknowledgement (message reached or not), retransmission (only when failed to deliver) and timeout. It can be made the multiple attempts of deliver messages. If the message gets lost in the way then the server can re-request for lost port.
- **Ordered -** In the TCP, the order of the messages are same as they send to the receiver.
- **Heavyweight -** TCP is fairly heavyweight and it needs the three packets to establish a socket connection, before any user data can sent. There are three packets are **SYN, SYN+ACK, and ACK.**

IP Addresses and Ports

IP addresses are the unique address which is used to identify a device over the internet. It is unique for every computer system. Port is an endpoint for the communication in an operating system. Let's understand in the simple way.

Example -

In the previous example, A wants to send letter to B. So A requires to know the address of B to deliver the package successfully. Now, A has the unique postal address of B and hence the postman delivers the letter successfully. So IP address is like the postal Address.

A system might have thousand running services but we can uniquely identify a service using **port number**. There are total of 0-65535 ports on a system.

Sometime the Port number can be seen in web or other uniform resource locators (URL) as well. By default, HTTP uses port 80 HTTPS port 443. Below is the example of the common ports.

Port Number	Description
22	Secure Shell
23	Telnet Remote Login Service
25	Simple Mail Transfer Protocol (SMTP)

53	Domain Name System (DNS) Service
80	Hyper Text Transfer Protocol (HTTP) which used in WWW.

There are two types of the IP addresses.

- **Private IP address -** It ranges from **168.0.0-192.168.255.255), 172.16.0.0-172.31.255.255)** or (**10.0.0.0-10.255.255.255**).
- **Public IP address -** A public IP address is an IP address that our home or business router receives from the internet service provider.

Firewall

A firewall is a security device that monitors the incoming and outgoing traffic and allows the valid data packets and block data packets based on a set of security rules. The firewall establishes a barrier between our internal network and incoming traffic from the external sources to block the corrupt traffic such as virus and hackers.

Firewall has the pre-define rules which used to filter the incoming traffic from the suspicious sources to stops attack.

For example -

Suppose the IP addresses as houses and port numbers as room within the house. Only authenticate people (source address) are allowed to enter the house (destination address).

Why Python for network programming?

There are many reasons that Python can be used in the network programming. Python is the most powerful language. It is a syntactically simple in comparison of other programming languages.

Python's libraries are available for almost everything. We can develop a website using the Python. We can make a face detection system using Python. We can use Python in testing also. We can use Python in the Machine fields such as data mining, data analysis, data modeling, etc. Python also used for network programming. It includes network supports libraries such as FTP, HTTP and so on.

Using the third-party libraries, we can create effective and user-friendly applications. There is proper documentation for the third party libraries; hence we can get help from it.

We can establish the connection between the two devices using the Python's socket module and create client-server applications. We can also create user-define class and use it as an application-layer protocols.

Python is powerful enough for socket programming and the perfect choice for networking programming.

Basic of Socket Programming

We have learned the basic concepts of the network and understand the basic network terminology. Before starting the Python network programming, we should go through the socket introduction.

The networking and sockets programming are the huge subjects. Computer network has many topics to explore but here we discuss the basic network programming using Python.

Let's understand the socket and see what they are and why we use them.

What is Socket?

First, we need to learn the internet connection. The internet connection is basically used for connect two end points across the internet for data sharing and other things. The internet connection provides the facility to the one process from computer C1 can communicate to a process from a computer C2. It consists of following features.

- **Reliable** - The data transfer in internet connection is safe.
- **Point-to-Point** - It establish the connection between 2 points.
- **Full-Duplex** - The full-duplex means the data transfer happens in two ways. The client send data to server and server send data to client as well.

Sockets are the endpoints of a two-directional, point-to-point communication channel. Sockets are generally used to transfer the message across the network. For example - An internet connection between client (web-browser) and server (www.javatpoint.com). Now, there is two endpoints **Client Socket** and a **Server Socket.**

Sockets were initially used in 1971 and later it turned into an API in the Berkeley Software Distribution (BSD) operating system which was introduced in the 1983 called Berkeley socket.

When the internet was invented along with the World Wide Web in 1990, the client-server applications came in extensive use. Among the various form of IPC (Internet Service Provider), Sockets are the by far most popular.

The client-server applications are the most common type of the socket applications. In this type of application, one side is a server and waits for establish the connection to the client. In this tutorial, we will cover this type of applications.

Socket API Functions

The socket module of Python offers an interface to the Berkeley sockets API. Below are the primary functions and methods of Python socket module.

- **socket() -** It is used to create the socket object.
- **bind() -** It is used to bind-address to the socket. It takes two arguments hostname and port number.
- **listen() -** It is used to establish and start the TCP listener.
- **accept() -** It is used to TCP client connection until the connection is established.
- **connect() -** It is used to initiate TCP server connection.
- **sendto() -** It is used to send the UDP messages.
- **send() -** It is used to send the TCP messages.
- **recv() -** It is used to receive the TCP messages.
- **close() -** It is used to close a socket.

Python socket API offers the sophisticated and user-friendly built-in functions that directly maps to the system calls. Python also has classes that can be used the low-level socket function easier. Python also provides the many modules that implement higher-level internet protocols such as HTTP and SMTP.

Creating a Socket

When we clicked on the link that brought us to a particular page. The web-browser acts as the following ways.

Client.py

```
import socket
# we create a socket object to establish the connection.
Client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# now connect to the web server on port 80
# - the normal http port
Client_socket.connect(("www.javatpoint.com", 80))
```

The above code will work on the client side. When the client tries to communicate the server, an **Ephermal Port** is assigned by the operating system for the connection. An Epherimal Port is nothing but a random port which is assigned by the operating system. The client socket closes as soon as data exchange is complete. The client socket is only used for the one time.

The server creates a server socket which is slightly complex than client socket. Let's see what happen at server side.

Server.py

```
import socket
Server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
#We bind the socket to a public host and a well-known port
Server_socket.bind((socket.gethostname(), 80))

#It become a server socket and listen for connections
Server_socket.listen(5)
```

Difference b/w Client socket and Server socket.

- Client sockets closes after the request is complete, where the server sockets are not short lived. For example - A popular video stream website YouTube, we might need a single request to the youtube.com has to be up 24*7 for any request which it might receive from user across the world.
- Client socket uses the Ephermal Port for connection, and server socket needs a standard or well defined port connection such as Port 80 for Normal HTTP Connection, Port 23 for Telnet etc.
- Client socket is an endpoint for communication, where server socket waits for request to come over the internet.

Python Socket Module

Python socket module provides the socket() method which is necessary to call while creating a socket. Below is the syntax of the socket() method.

1. Create_Socket = socket.socket (socket_family, socket_type, protocol = 0)

- **socket_family -** It can be either AF_UNIX or AF_INET. We will explain only INET socket in this tutorial, because they are at least 99% socket in use.
- **socket_type -** It can be moreover **SOCK_STREAM** or **SOCK_DGRM**.
- **Protocol -** By default, it assigns to 0.

In the previous section, we have discussed the client-server socket module. Now we will learn its methods and their use.

Client Sockets Methods

The client socket method is given below.

connect()

This function is used to set up a connect to a remote socket at an address. An address format contains the host and port pair which is used for **AF_INET**address family.

Server Socket Methods

The server socket methods are given below.

bind()

This method is used to bind the socket to an address. The address's format depends on socket family mentioned above (AF_INET).

listen(backlog)

This method is used to listen for the connection made to the socket. The backlog is a term which used to represent the maximum number of the queued connection that must be listened before rejecting the connection.

accepts()

The accepts() method accepts a connection. The socket should be bound to an address and ready to listen the connections. It returns the **pair(conn,**

address)where **con** is a new socket object which can be used
to **send** and **receive** data on the connection, and **address** is the address linked to the socket on the other end of the connections.

Few Common Socket Methods

A few commonly used functions for server object is given below.

1. Server_Object = socket.socket(socket_family, socket_type, protocol = 0)

TCP Socket Methods	UDP Socket Methods
Server_Object.recv() - Receives TCP messages	**Server_Object.recvfrom()** - Receive UDP messages.
**Server_Object.send(**) - Transmit TCP messages	**Server_Object.sendto()** - Transmits UDP messages.

Some Basic Socket Methods

The following are the some important sockets methods.

- **close() -** The close() function is used to close the connection.
- **gethostname() -** This method is used to return a string that contain the hostname of the machine. The Python interpreter is currently executing. In that particular place. For example **- localhost**.
- **gethostbyname() -** The gethostname() method is used to know the current machine's IP address.

Types of Socket

The sockets are the two types; SOCK_STREAM and SOCK_DGRAM. We have a comparison of both sockets in the following table.

SOCK_STREAM	SOCK_DGRAM
It is used for TCP protocols	It is used for UDP protocols
It has reliable delivery	It has unreliable delivery
It provides guaranteed correct ordering of packets.	There is no order guaranteed.
It is connection-oriented	There is no notion of connection(UDP)
It is bidirectional	It is not Bidirectional

TCP Sockets

In this section, we will create the socket object using **socket.socket()** function and specify the socket type as **socket.SOCK_STREAM**. As we know that, A protocol is essential to send data one end to another. Here the TCP (Transmission Control Protocol) is used default protocol. It is highly operative and reliable. We can consider this protocol because -

- **It is reliable -** If the packet is missed in between the transfer to the other end then it retransmitted by the sender.
- **In order data deliver -** The message is send in the same order as it written by the sender.

On the other side, User Datagram Protocol (UDP) sockets make with socket.SOCK_DGRAM isn't reliable. The data read by the receiver can be same order as sender's writes.

Let's understand the following example of simple Client-Server Program.

Simple Server Program

Example -

```
#!/usr/bin/python

#This is tcp_server.py script

# Import socket module
import socket

# create a socket object
s = socket.socket()
# Get current machine name
host = socket.gethostname()
# Get port number for connection
port = 9999
# bind with the address
s.bind((host,port))
# listen for connections
print("Waiting for connection...")
s.listen(5)
```

```
# connect and accept from client
while True:
   conn,addr = s.accept()
   print('Got Connection from', addr)
   conn.send('Responce From the Server')
   # Close the connection
   conn.close()
```

The above code will not do anything as of now. It waits for a client to connect at the specified port. It will give the following output, if there is no error in the program.

Similarly, every site that we visit consists of a server on which it is hosted.

Now, we will create a ***client.py*** program to connect with the ***server.py*** file.

Simple Client Program

Consider the following ***client.py*** program. The client tries to set the connection to server's port; we are assigning the 9999 well well-defined port.

Example -

```
#!/usr/bin/python

#This is tcp_client.py script

import socket
#Create a socket
s = socket.socket()
# Get current machine name
host = socket.gethostname()
# Client wants to connect to server's
# port number 9999
port = 9999
# 1024 is buffer size or max amount
# of data to be received at once
s.connect((host,port))
print(s.recv(1024))
```

To get the result, first run the ***server.py*** file and then run ***client.py*** file. If there is no error then, it will give the following output.

Note - Here, the client and server files are running on the same machine, but in the real life server is situated on the different place. The point is to be notice here the client.py is terminated but the server.py is still running. This is also happen in real life scenario.

For example - When you make request to the javatpoint.com then it fulfill the request and its server continuously running (24*7) in the background.

Python Internet Module

The following are the list of the Python's module related to the network programming.

Protocol	Common function	Port No	Python module
HTTP	Web pages	80	httplib, urllib, xmlrpclib
NNTP	Usenet news	119	nntplib
FTP	File transfers	20	ftplib, urllib
SMTP	Sending email	25	smtplib
POP3	Fetching email	110	poplib
IMAP4	Fetching email	143	imaplib
Telnet	Command lines	23	telnetlib
Gopher	Document transfers	70	gopherlib, urllib

Above mentioned list contains all Python network module which frequently used to work over the network.

Working with UDP Sockets

We have learned that, if we don't mention the socket family and socket type then by default it is TCP. To create the UDP socket, we need to specify the socket family and socket type explicitly. Let's understand the following syntax.

Syntax -

```
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

Let's understand the following UDP server program.

UDP Server Program

Write the below script and save it named server.py

Example -

```
import socket

# It is used for UDP protocol
udp_socket = socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
# Host IP
udp_host = socket.gethostname()
# We are specifying port to connect
udp_port = 12345

#print type(udp_sock)  'type' can be used to see type
        # of any variable ('sock' here)

udp_socket.bind((udp_host,udp_port))

while True:
  print("Waiting for client...")
  data,addr = sock.recvfrom(1024)          #receive data from client
  print("Received Messages:",data," from",addr)
```

Output:

UDP Client Program

Save this file named udpclient.py

Example -

```
import socket
# For creating the udp socket
udp_socket = socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
# Host IP
udp_host = socket.gethostname()
# We are specifying port to connect
```

```
udp_port = 12345

msg = "Welcome to JavaTpoint"
print("UDP target IP:", udp_host)
print("UDP target Port:", udp_port)
# Sending message to UDP server
udp_socket.sendto(msg,(udp_host,udp_port))
```

Output:

```
Waiting for client....

Received Messages: Welcome To JavaTpoint from('192.168.43.217,5342')

Waiting for client
```

We have discussed both sockets and establish the connection between the client-server sockets.

Echo Client and Server

An echo server is an application which permits a client to fetch information from server. The client can transmit the message or request to the server and the server can receive the request and send, return back to the client, or echo it.

Working with Input and Output

An echo server can receive a client message and recurrences it back to the client. To receiving the input and output stream, we need to develop the communication channels between the client and server. These input/output streams are movements of the data that means the client can send message to the server and the server can resonance back the message to the client.

Let's understand the following example:

Echo Server

```
import socket
# Standard loopback interface address (localhost)
host_name = '127.0.0.1'
# Specified Port to listen on (non-privileged ports are > 1023)
port_name = 65432
```

```
# TCP Sockets
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((host_name, port_name))
    s.listen()
    conn, addr = s.accept()
    with conn:
        print('Connected by', addr)
        while True:
            data = conn.recv(1024)
            if not data:
                break
            conn.sendall(data)
```

Explanation -

The **socket.socket()** function creates a socket object that supports the **context manager type,** we can also use it in a **with** statement and don't need to call **close()** function.

We have mentioned the address family and **socket type.AF_INET** that is an Internet address family for **IPv4.** The SOCK_STREAM is the socket type of TCP protocol to transport our message in the network.

We have used the **bind()** function that used to connect the socket with the explicit network interface and port number.

```
s.bind((host_name, port_name))
```

Here, the **host_name** represents the **hostname**, IP address, **or empty string**. The host must be IPv4-formatted address string.

The **port** number be an integer number which lies in between 1 to 65535 (0 is reserved). It denotes the TCP port number to accept connection from clients. Some system may require the superuser authentication if port is < 1024.

Next, we have used the **listen()** function, which enables a server to **accept()**connections.

```
s.listen()
conn, addr = s.accept()
```

By default, the **listen()** function consists of ***backlog*** parameter.
The ***backlog***parameter represents the number of rejected connection that system will allows before **declining** new connection.

The backlog helps to manage the maximum length of the length of the queue for pending connection.

When a client establish the connection, it returns a new socket object using the **accept()** function.

Now, we have a socket object from accept() function. It is important to recall that the socket that we will use to communicate must be distinct from the listening socket that the server is using to accept new connections.

```
with conn:
      print('Connected by', addr)
      while True:
         data = conn.recv(1024)
         if not data:
            break
         conn.sendall(data)
```

Once we complete the process to get the conn object from **accept()** function. We used the infinite while loop over blocking calls to **conn.recv()**. The client sends and echoes it back using **conn.sendall().**

Echo Client

We are describing the echo client program and save it echo-client.py:

Program -

```
import socket
# This represent the server's hostname or IP address
HOST = '127.0.0.1'
# This is port number used by the server
PORT = 65432

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
   s.connect((HOST, PORT))
   s.sendall(b'Hello, world')
```

```
data = s.recv(1024)

print('Received', repr(data))
```

Explanation:

The echo client is much simple in comparison of echo server. It links a socket object, connects to the server and calls **s.sendall()** to send messages.
The **s.recv()** to read the server reply and we can print it.

Blocking and Non-Blocking Sockets

In the previous sections, we learned the client sends the request to the server and server process the request; send back a response using the sockets (TCP/UDP).

Blocking Socket I/O

TCP sockets are located in a blocking mode by default. It means the control is not transfer to our program until some specific operation is complete. Let's see the example -

If we call **connect()** method, the connection blocks the program until the task is complete. Let's understand another example. When we write a program for web browser client that will connect to the web server, it is necessary to use the stop functionality that can be used to interrupt an active connection process the intermediate of its operations at any place. We can achieve it by **assigning** the socket in the **non-blocking mode.**

Non - Blocking Socket I/O

The **setblocking(1)** function is used to set the blocking
and **setblocking(0)**function is used to unset the block. Let's understand the following example of **Blocking Socket**.

Save file blockclient.py

Program

```
import socket
# Creating a socket object
```

```
socket_obj = socket.socket()

host = socket_obj.gethostname()
socket_obj.connect((host, 12345))
socket_obj.setblocking(1)

# Or simply omit this line as by default TCP sockets
# are in blocking mode
# Huge amount of data to be sent
data = "Hello Python\n" *10*1024*1024
# Send data till true
assert socket_obj.send(data)
```

Now, understand the following blockserver.py:

Program

```
import socket
# Creating a socket object
socket_obj = socket.socket()

# Defining host name
host = socket.gethostname()
# defining port number
port = 12345

socket_obj.bind((host, port))
socket_obj.listen(5)

while True:
    # accept the connection
    conn, addr = socket_obj.accept()

    data = conn.recv(1024)
    # till data is coming
    while data:
        print(data)
        data = conn.recv(1024)
        # It will execute when all data is received
    print("All Data Received")
    conn.close()
```

```
break
```

First, run the blockServer.py file and then blockClient.py. The server will continuously printing the message **Hello JavaTpoint.** This will keep on until all the data is sent. In the above code, the client will not be printed the **Hello JavaTpoint** for long time. It will take much time due to client has to send the large number of string. It will happen until the socket input/output will get blocked.

The **send()** function is used to convey the data all over the server, while the write buffer will get complete. The kernel will hold the process to the sleep until the data in the buffer is transported to the client and the buffer is empty again. Once the buffer is empty, the kernel will start its process up again to get the next chunk of data that is to be transported.

Now consider a Non - Blocking Socket.

Program

```
# non_blocking_client.py

import socket

socket_object = socket.socket()

host = socket.gethostname()
socket_object.connect((host, 12345))
# Now setting to non-blocking mode

socket_object.setblocking(0)
# Huge amount of data to be sent
data = "Hello Python\n" * 10 * 1024 * 1024
# Send data till true
assert socket_object.send(data)
```

When we run the **non_block_client.py**, the program will run for a small time, it will print the "All Data Received" and quickly finished.

If we create the socket non-blocking by the **setblocking(0),** It will never wait for the operation to complete. So when we call the send() function it will try to put maximum amount of data in the buffer.

Closing a Connection

Generally, we use the ***shutdown*** before on a socket before closing it. The close is the same as shutdown() method. So we don't need to call the **shutdown()**explicitly.

We can also use the **shutdown()** function effectively using the HTTP-like exchange. If the client use the **shutdown(1)** that indicates the "The client is done sending, but can still receive." The server can detect "EOF" by a receive of 0 bytes. The server can predict, it has complete request and sends a reply. If the send completes successfully then, indeed, the client was still receiving.

In Python, The automatic shutdown takes a step ahead, means the socket is garbage collected, it will do a close if it's needed. But this is the bad habit to use. If the socket terminates without doing a close. The socket at the other hand may the other socket is too slow to send the request. So it is necessary to close the socket once you have completed the job.

The Tornado Framework

The Tornado framework is one of the most popular frameworks for networking programming in Python. It consists of wide range of network support libraries. Let's discuss how to use it to build **WebSockets**.

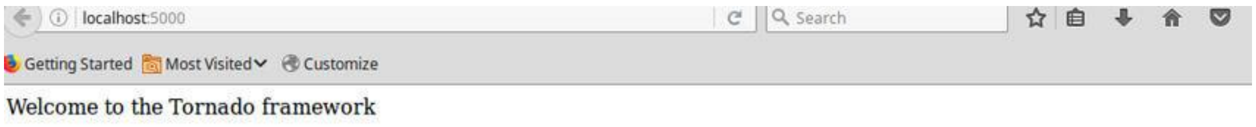
Tornado is an asynchronous networking library and web framework. The Tornado uses the non-blocking I/O, and hence capable to handle the tens of thousands of open connections. This characteristic makes it perfect for long polling, **WebSockets,** and other applications that need a long-lived connection to each user.

Let's understand the following example of simple Tornado **WebSocket**.

Example -

```
import tornado.ioloop
import tornado.web


class ApplicationHandler(tornado.web.RequestHandler):

    def get(self):
        self.message = message = """<html>
<head>
  <title>Tornado Framework</title>

</head>
<body
  <h2>Welcome to the Tornado framework</h2>
</body>
</html>"""
        self.write(message)


if __name__ == "__main__":
  application = tornado.web.Application([
      (r"/", ApplicationHandler),
  ])
  application.listen(5001)
  tornado.ioloop.IOLoop.instance().start()
```

Output:



Welcome to the Tornado framework

Explanation:

In the above code,

- We have defined the class **ApplicationHandler** which uses as the handler for request and return a response using the **write()**
- The **main()** method is the entrance of the program.
- The class **web.Application** creates the base for the web application and accepts a collection of handlers.
- The Application listens on port 5000, and a client can communicate to this application using the same port.

- The **ioloop.IOLoop.instance().start()** function is used to create a nonblocking thread for an application.

In this article, we have discussed the basics concepts of the network programming using the Python. We have defined the basic network terminologies and create the simple server and client. The Networking programming is vast field and hard to cover in the single tutorial but we are tried to cover all important concepts regarding network programming using Python.
