Network Programming Paradigm

Introduction

The Network paradigm involves thinking of computing in terms of a client, who is essentially in need of some type of information, and a server, who has lots of information and is just waiting to hand it out. Typically, a client will connect to a server and query for certain information. The server will go off and find the information and then return it to the client.

In the context of the Internet, clients are typically run on desktop or laptop computers attached to the Internet looking for information, whereas servers are typically run on larger computers with certain types of information available for the clients to retrieve. The Web itself is made up of a bunch of computers that act as Web servers; they have vast amounts of HTML pages and related data available for people to retrieve and browse. Web clients are used by those of us who connect to the Web servers and browse through the Web pages.

Network programming uses a particular type of network communication known as sockets. A socket is a software abstraction for an input or output medium of communication.

What is Socket?

- A socket is a software abstraction for an input or output medium of communication.
- Sockets allow communication between processes that lie on the same machine, or on different machines working in diverse environment and even across different continents.
- A socket is the most vital and fundamental entity. Sockets are the end-point of a two-way communication link.
- An endpoint is a combination of IP address and the port number.
- range of port numbers from 0 to 65535

For Client-Server communication,

- Sockets are to be configured at the two ends to initiate a connection,
- Listen for incoming messages
- Send the responses at both ends
- Establishing a bidirectional communication.

Socket Types

Datagram Socket

- A datagram is an independent, self-contained piece of information sent over a network whose arrival, arrival time, and content are not guaranteed. A datagram socket uses User Datagram Protocol (UDP) to facilitate the sending of datagrams (self-contained pieces of information) in an unreliable manner. Unreliable means that information sent via datagrams isn't guaranteed to make it to its destination.

Stream Socket:

- A stream socket, or connected socket, is a socket through which data can be transmitted continuously. A stream socket is more akin to a live network, in which the communication link is continuously active. A stream socket is a "connected" socket through which data is transferred continuously.

Socket in Python

```
sock_obj = socket.socket( socket_family, socket_type, protocol=0)
```

socket_family: - Defines family of protocols used as transport mechanism.

Either AF_UNIX, or

AF_INET (IP version 4 or IPv4).

socket_type: Defines the types of communication between the two end-points.

SOCK_STREAM (for connection-oriented protocols, e.g., TCP), or

SOCK_DGRAM (for connectionless protocols e.g. UDP).

protocol: We typically leave this field or set this field to zero.

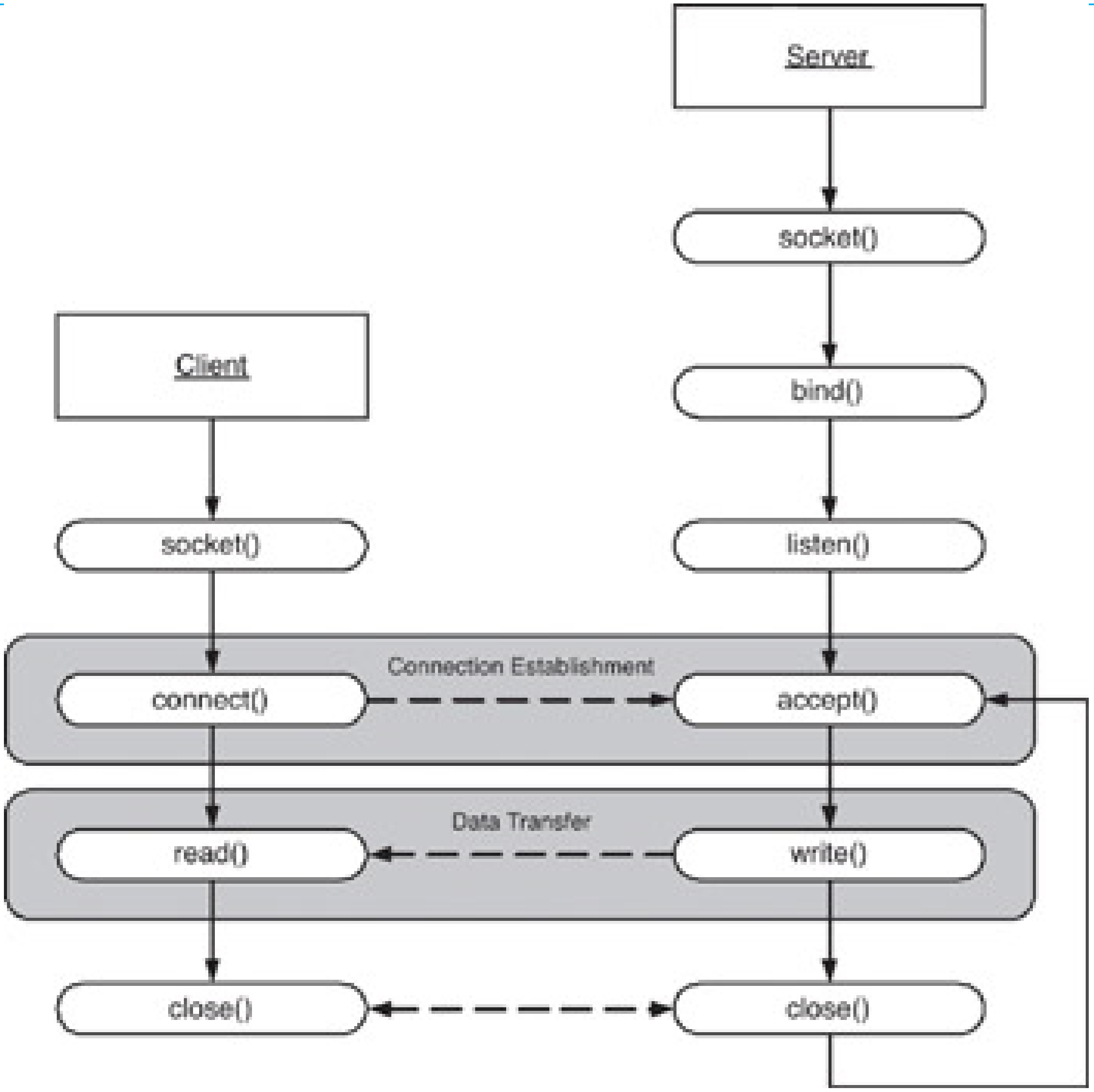
Example:

```
#Socket client example in python
import socket
#create an AF_INET, STREAM socket (TCP)
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
print 'Socket Created'
```

Socket Creation

```
import socket

import sys

try:

        #create an AF_INET, STREAM socket (TCP)

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

except socket.error, msg:

        print 'Failed to create socket. Error code: ' + str(msg[0]) + ' , Error message : ' + msg[1]

        sys.exit();


print 'Socket Created'
```

Client/server symmetry in Sockets applications



Socket in Python

To create a socket, we must use socket.socket() function available in the Python socket module, which has the general syntax as follows:

S = socket.socket(socket_family, socket_type, protocol=0)

socket_family: This is either AF_UNIX or AF_INET. We are only going to talk about INET sockets in this tutorial, as they account for at least 99% of the sockets in use.

socket_type: This is either SOCK_STREAM or SOCK_DGRAM.

Protocol: This is usually left out, defaulting to 0.

Client Socket Methods

Following are some client socket methods:

connect( ) : To connect to a remote socket at an address. An address format(host, port) pair is used for AF_INET address family.

Socket in Python

Server Socket Methods

bind( ): This method binds the socket to an address. The format of address depends on socket family mentioned above(AF_INET).

listen(backlog) : This method listens for the connection made to the socket. The backlog is the maximum number of queued connections that must be listened before rejecting the connection.

accept( ) : This method is used to accept a connection. The socket must be bound to an address and listening for connections. The return value is a pair(conn, address) where conn is a new socket object which can be used to send and receive data on that connection, and address is the address bound to the socket on the other end of the connection.

General Socket in Python

sock_object.recv():

Use this method to receive messages at endpoints when the value of the protocol parameter is TCP.

sock_object.send():

Apply this method to send messages from endpoints in case the protocol is TCP.

sock_object.recvfrom():

Call this method to receive messages at endpoints if the protocol used is UDP.

sock_object.sendto():

Invoke this method to send messages from endpoints if the protocol parameter is UDP.

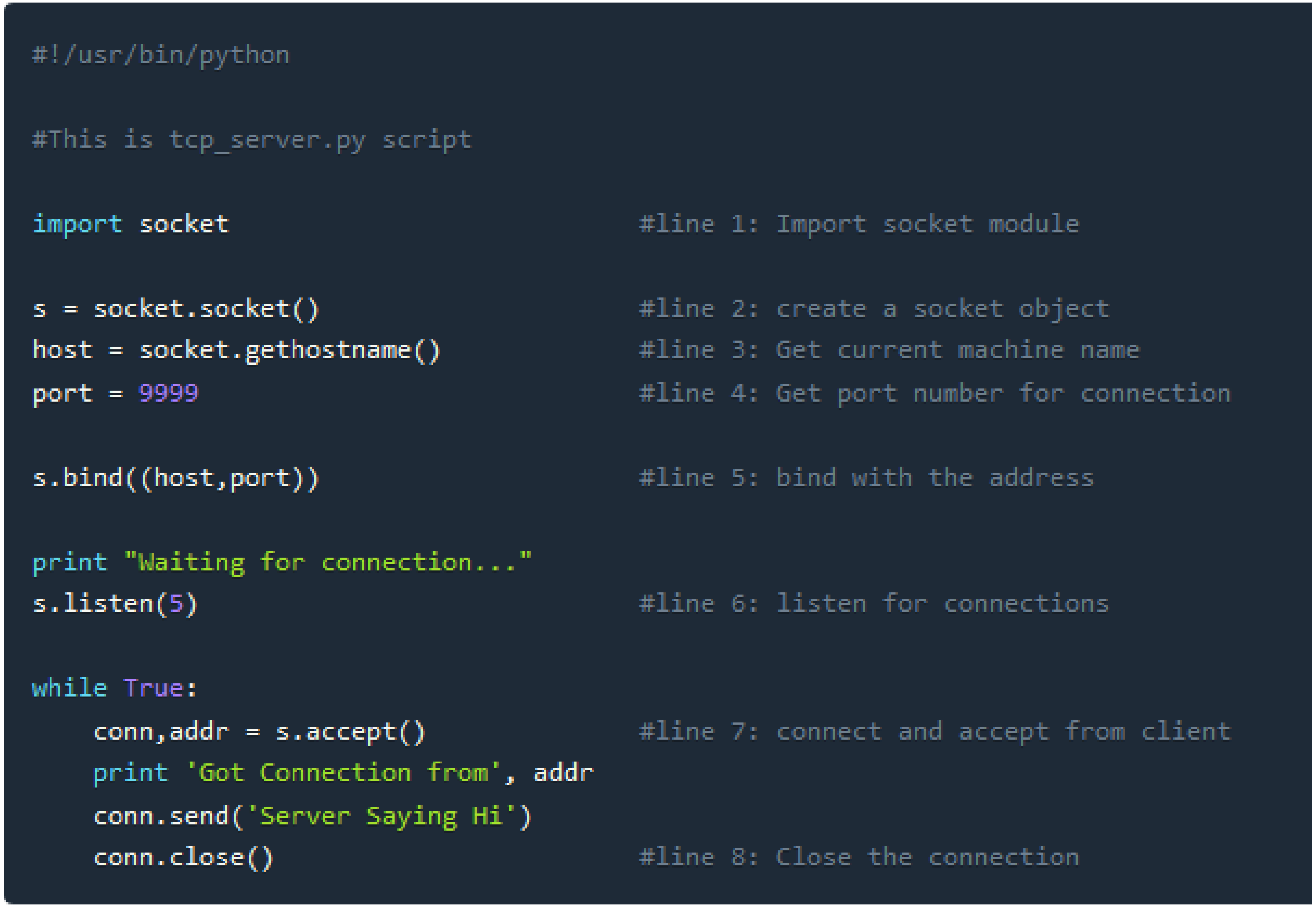
sock_object.gethostname():

This method returns hostname.

sock_object.close():

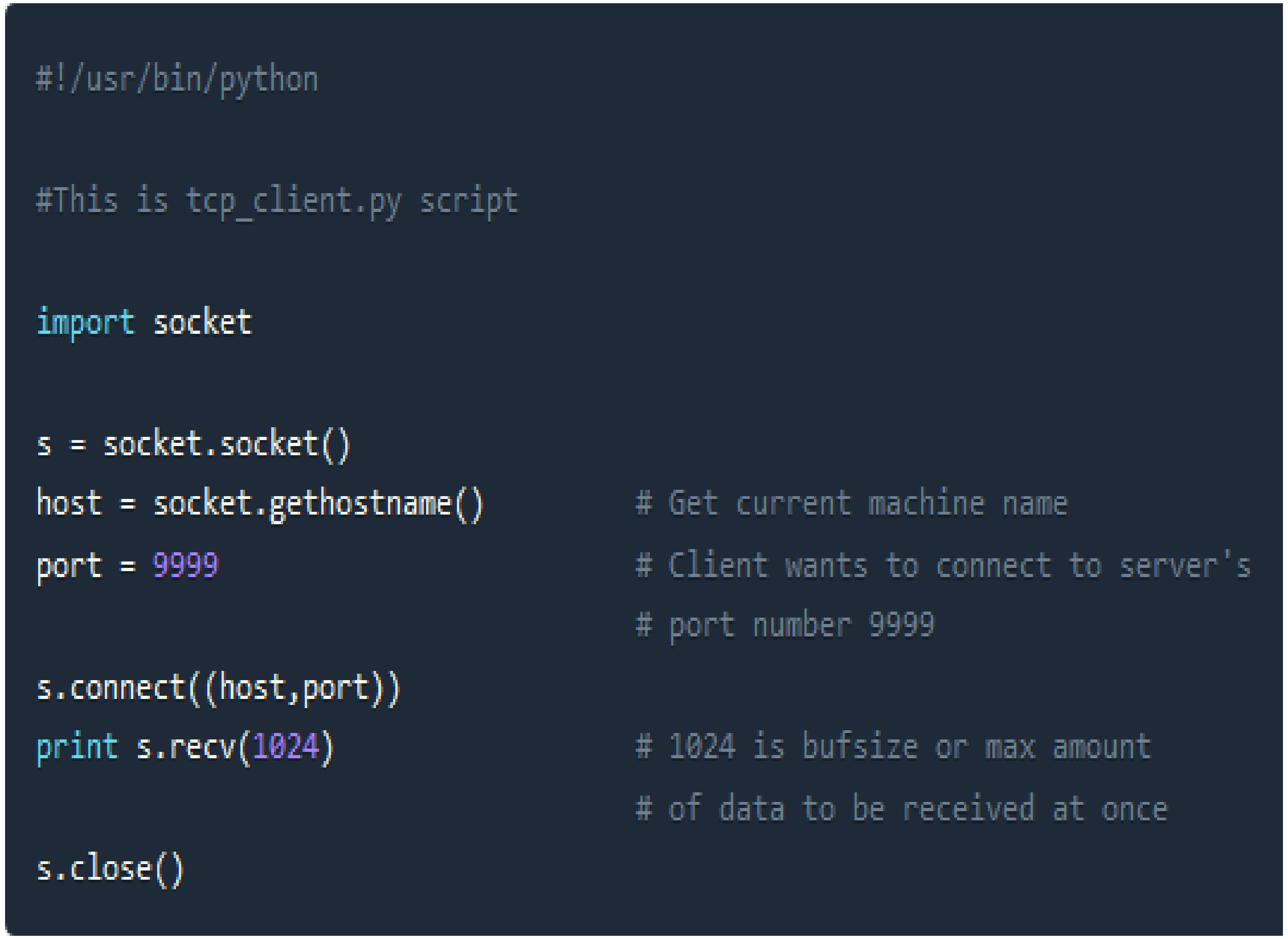
This method is used to close the socket. The remote endpoint will not receive data from this side.

Simple TCP Server



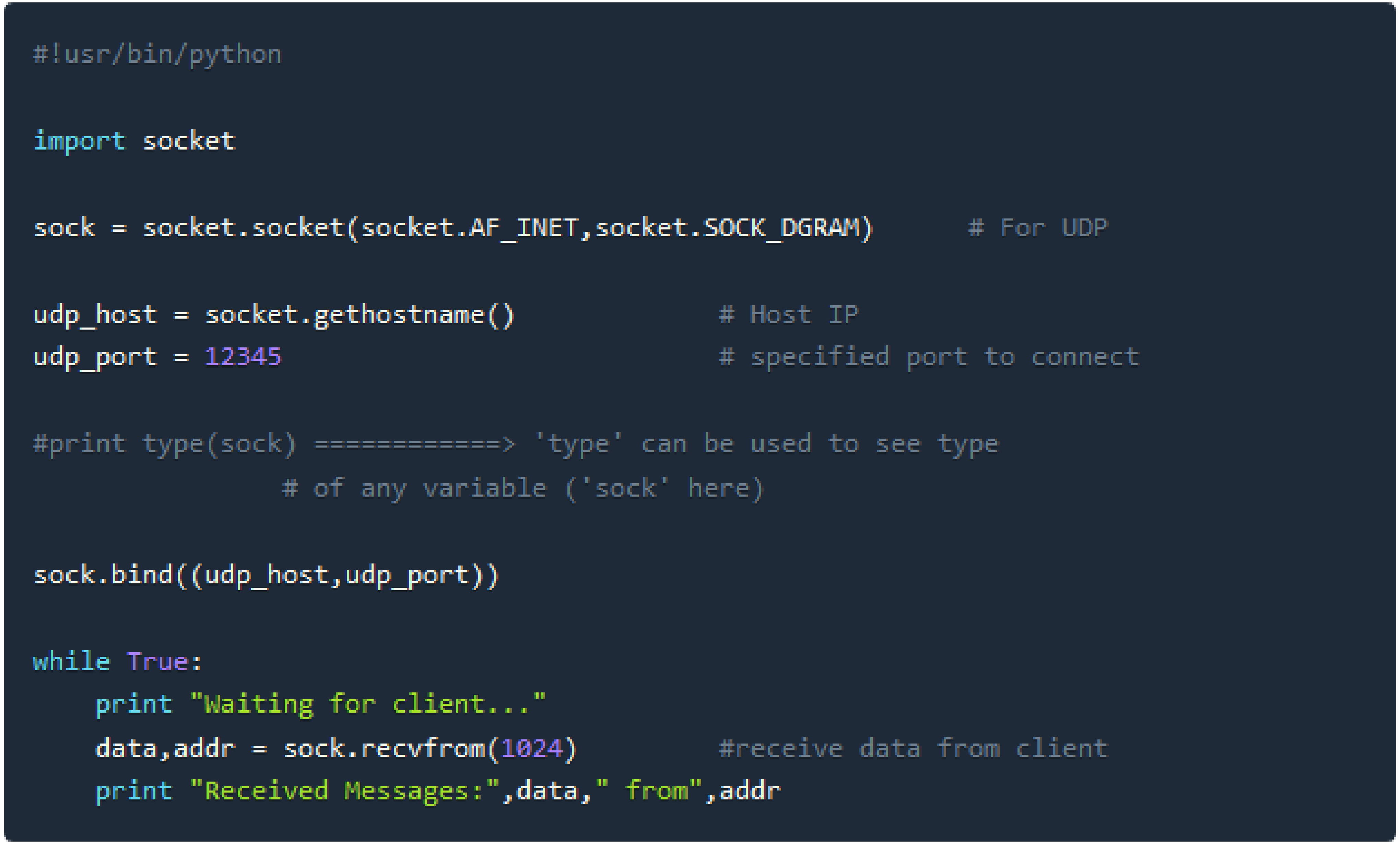
```python
#!/usr/bin/python

#This is tcp_server.py script

import socket                           #line 1: Import socket module

s = socket.socket()                     #line 2: create a socket object
host = socket.gethostname()             #line 3: Get current machine name
port = 9999                             #line 4: Get port number for connection

s.bind((host,port))                     #line 5: bind with the address

print "Waiting for connection..."
s.listen(5)                             #line 6: listen for connections

while True:
    conn,addr = s.accept()              #line 7: connect and accept from client
    print 'Got Connection from', addr
    conn.send('Server Saying Hi')
    conn.close()                        #line 8: Close the connection
```

Simple TCP Client



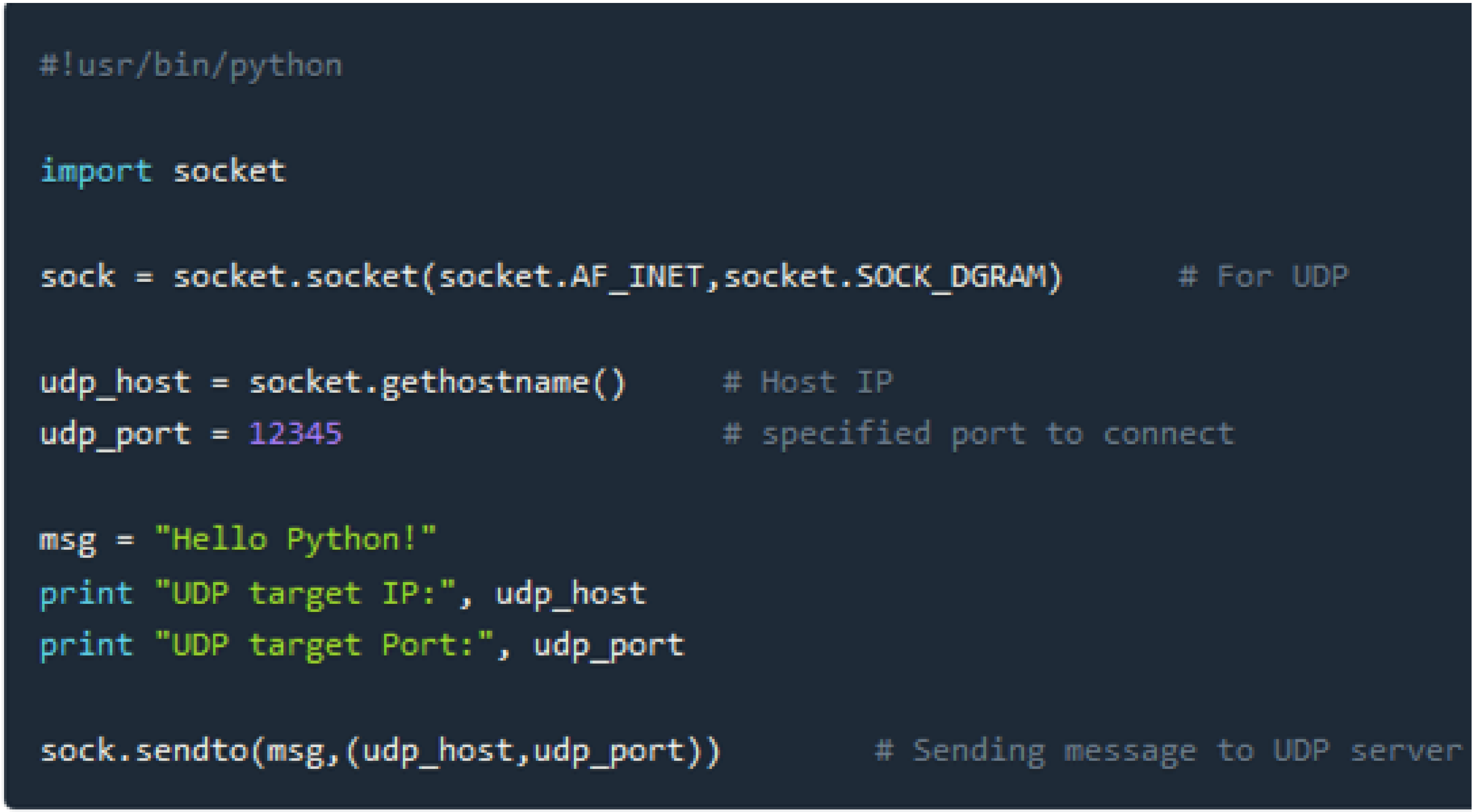
```
#!/usr/bin/python

#This is tcp_client.py script

import socket

s = socket.socket()
host = socket.gethostname()         # Get current machine name
port = 9999                         # Client wants to connect to server's
                                    # port number 9999

s.connect((host,port))
print s.recv(1024)                  # 1024 is bufsize or max amount
                                    # of data to be received at once

s.close()
```

Simple UDP Server



```python
#!usr/bin/python

import socket

sock = socket.socket(socket.AF_INET,socket.SOCK_DGRAM)      # For UDP

udp_host = socket.gethostname()             # Host IP
udp_port = 12345                            # specified port to connect

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

sock.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
```

Simple UDP Client



```
#!usr/bin/python

import socket

sock = socket.socket(socket.AF_INET,socket.SOCK_DGRAM)      # For UDP

udp_host = socket.gethostname()         # Host IP
udp_port = 12345                        # specified port to connect

msg = "Hello Python!"
print "UDP target IP:", udp_host
print "UDP target Port:", udp_port

sock.sendto(msg,(udp_host,udp_port))        # Sending message to UDP server
```

```
# An example script to connect to Google using socket
# programming in Python
import socket # for socket
import sys
try:
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    print "Socket successfully created"
except socket.error as err:
    print "socket creation failed with error %s" %(err)
# default port for socket
port = 80

try:
    host_ip = socket.gethostbyname('www.google.com')
except socket.gaierror:

    # this means could not resolve the host
    print "there was an error resolving the host"
    sys.exit()

# connecting to the server
s.connect((host_ip, port))

print "the socket has successfully connected to google \
on port == %s" %(host_ip)
```

Output :

```
Socket successfully created the socket has successfully connected to google on port ==
173.194.40.19
```

A simple server-client program

- **Server** :
  A server has a bind() method which binds it to a specific ip and port so that it can listen to incoming requests on that ip and port.
- A server has a listen() method which puts the server into listen mode. This allows the server to listen to incoming connections.
- And last a server has an accept() and close() method.
- The accept method initiates a connection with the client and the close method closes the connection with the client.

```
# first of all import the socket library
import socket
# next create a socket object
s = socket.socket()
print "Socket successfully created"
# reserve a port on your computer in our
# case it is 12345 but it can be anything
port = 12345
# Next bind to the port
# we have not typed any ip in the ip field
# instead we have inputted an empty string
# this makes the server listen to requests
# coming from other computers on the network
s.bind(('', port))
print "socket binded to %s" %(port)
# put the socket into listening mode
```

```
print "socket is listening"

# a forever loop until we interrupt it or
# an error occurs
while True:

# Establish connection with client.
c, addr = s.accept()
print 'Got connection from', addr

# send a thank you message to the client.
c.send('Thank you for connecting')

# Close the connection with the client
c.close()
```

Output :

- # in the server.py terminal
- Socket successfully created
- socket binded to 12345
- socket is listening
- Got connection from ('127.0.0.1', 52617)

Now for the client side:

```
# Import socket module
import socket

# Create a socket object
s = socket.socket()

# Define the port on which you want to connect
port = 12345

# connect to the server on local computer
s.connect(('127.0.0.1', port))

# receive data from the server
print s.recv(1024)
# close the connection
s.close()
```

```
# start the server:
$ python server.py
Socket successfully created
socket binded to 12345
socket is listening
Got connection from ('127.0.0.1', 52617)
# start the client:
$ python client.py
Thank you for connecting
```

Other Languages : Powershell

- Classes can be defined in functionality
- DSC enhancements
- Transcriptions available in all hosts
- Major enhancements to debugging, including the ability to debug Windows PowerShell jobs
- Network switch module
- OneGet for managing software packages
- PowerShellGet for managing Windows PowerShell modules through OneGet
- Performance gain when using COM objects

Bash:

- Directory manipulation, with the **pushd**, **popd**, and **dirs** commands.
- Job control, including the **fg** and **bg** commands and the ability to stop jobs with CTRL-Z.
- Brace expansion, for generating arbitrary strings.
- Tilde expansion, a shorthand way to refer to directories.
- Aliases, which allow you to define shorthand names for commands or command lines.
- Command history, which lets you recall previously entered commands.

TCL O

- Cross Platform Portability. Runs on Windows, Mac OS X, Linux, and virtually every variant **of** unix.
- Event driven programming. Trigger events based on variable read / write / unset. ...
- Object Oriented Programming. Mixins. ...
- Simple Grammar.
- Full unicode support. ...
- Flexible. ...
- Powerful introspection capabilities. ...
- Library interface.