## Python Socket programming

Python provides two levels of access to network services. At a low level, you can access the basic socket support in the underlying operating system, which allows you to implement clients and servers for both connection-oriented and connectionless protocols.

Python also has libraries that provide higher-level access to specific application-level network protocols, such as FTP, HTTP, and so on.

This tutorial gives you understanding on most famous concept in Networking - Socket Programming

What is Sockets?

Sockets are the endpoints of a bidirectional communications channel. Sockets may communicate within a process, between processes on the same machine, or between processes on different continents.

Sockets may be implemented over a number of different channel types: Unix domain sockets, TCP, UDP, and so on. The socket library provides specific classes for handling the common transports as well as a generic interface for handling the rest.

Sockets have their own vocabulary:

**Term Description**

*domain* The family of protocols that will be used as the transport mechanism. These values are constants such as AF\_INET, PF\_INET, PF\_UNIX, PF\_X25, and so on.

*type* The type of communications between the two endpoints, typically SOCK\_STREAM for connection-oriented protocols and SOCK\_DGRAM for connectionless protocols.

*protocol* Typically zero, this may be used to identify a variant of a protocol within a domain and type.

*hostname* The identifier of a network interface:

A string, which can be a host name, a dotted-quad address, or an IPV6 address in colon (and possibly dot) notation

A string "<broadcast>", which specifies an INADDR\_BROADCAST address.

A zero-length string, which specifies INADDR\_ANY, or

An Integer, interpreted as a binary address in host byte order.

*port* Each server listens for clients calling on one or more ports. A port may be a Fixnum port number, a string containing a port number, or the name of a service.

The socket Module:

To create a socket, you must use the *socket.socket()* function available in socket module, which has the general syntax:

*s = socket.socket (socket\_family, socket\_type, protocol=0)*

Here is the description of the parameters:

socket\_family: This is either AF\_UNIX or AF\_INET, as explained earlier.

socket\_type: This is either SOCK\_STREAM or SOCK\_DGRAM.

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

Once you have socket object, then you can use required functions to create your client or server program. Following is the list of functions required:

**Server Socket Methods:**

Method Description

s.bind() This method binds address (hostname, port number pair) to socket.

s.listen() This method sets up and start TCP listener.

s.accept() This passively accept TCP client connection, waiting until connection arrives (blocking).

Client Socket Methods:

Method Description

s.connect() This method actively initiates TCP server connection.

General Socket Methods:

Method Description

s.recv() This method receives TCP message

s.send() This method transmits TCP message

s.recvfrom() This method receives UDP message

s.sendto() This method transmits UDP message

s.close() This method closes socket

socket.gethostname() Returns the hostname.

**A Simple Server:**

To write Internet servers, we use the socket function available in socket module to create a socket object. A socket object is then used to call other functions to setup a socket server.

Now call bind(hostname, port function to specify a port for your service on the given host.

Next, call the accept method of the returned object. This method waits until a client connects to the port you specified, and then returns a connection object that represents the connection to that client.

*#!/usr/bin/python # This is server.py file*

*import socket # Import socket module*

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

*host = socket.gethostname() # Get local machine name*

*port = 12345 # Reserve a port for your service.*

*s.bind((host, port)) # Bind to the port*

*s.listen(5) # Now wait for client connection.*

*while True:*

*c, addr = s.accept() # Establish connection with client.*

*print 'Got connection from', addr*

*c.send('Thank you for connecting')*

*c.close() # Close the connection*

**A Simple Client:**

Now we will write a very simple client program which will open a connection to a given port 12345 and given host. This is very simple to create a socket client using Python's socket module function.

The *socket.connect(hosname, port )* opens a TCP connection to hostname on the port. Once you have a socket open, you can read from it like any IO object. When done, remember to close it, as you would close a file.

The following code is a very **simple client** that connects to a given host and port, reads any available data from the socket, and then exits:

*#!/usr/bin/python # This is client.py file*

*import socket # Import socket module*

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

*host = socket.gethostname() # Get local machine name*

*port = 12345 # Reserve a port for your service.*

*s.connect((host, port))*

*print s.recv(1024)*

*s.close # Close the socket when done*

Now run this server.py in background and then run above client.py to see the result.

# Following would start a server in background.

$ python server.py &

# Once server is started run client as follows:

$ python client.py

This would produce following result:

*Got connection from ('127.0.0.1', 48437)*

*Thank you for connecting*

**Python Internet modules**

A list of some important modules which could be used in Python Network/Internet 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

Please check all the libraries

## Python Multithreading

Running several threads is similar to running several different programs concurrently, but with the following benefits:

Multiple threads within a process share the same data space with the main thread and can therefore share information or communicate with each other more easily than if they were separate processes.

Threads sometimes called light-weight processes and they do not require much memory overhead; theycare cheaper than processes.

A thread has a beginning, an execution sequence, and a conclusion. It has an instruction pointer that keeps track of where within its context it is currently running.

It can be pre-empted (interrupted)

It can temporarily be put on hold (also known as sleeping) while other threads are running - this is called yielding.

**Starting a New Thread:**

To spawn another thread, you need to call following method available in thread module:

*thread.start\_new\_thread ( function, args[, kwargs] )*

This method call enables a fast and efficient way to create new threads in both Linux and Windows.

The method call returns immediately and the child thread starts and calls function with the passed list of agrs. When function returns, the thread terminates.

Here args is a tuple of arguments; use an empty tuple to call function without passing any arguments. kwargs is an optional dictionary of keyword arguments.

**(1)**

Although it is very effective for low-level threading, but the thread module is very limited compared to the newer threading module.

The Threading Module:

The newer threading module included with Python 2.4 provides much more powerful, high-level support for threads than the thread module discussed in the previous section.

The threading module exposes all the methods of the thread module and provides some additional methods:

*threading.activeCount():* Returns the number of thread objects that are active.

*threading.currentThread():* Returns the number of thread objects in the caller's thread control.

*threading.enumerate():* Returns a list of all thread objects that are currently active.

In addition to the methods, the threading module has the Thread class that implements threading. The methods provided by the Thread class are as follows:

*run()*: The run() method is the entry point for a thread.

*start()*: The start() method starts a thread by calling the run method.

*join([time])*: The join() waits for threads to terminate.

*isAlive()*: The isAlive() method checks whether a thread is still executing.

*getName()*: The getName() method returns the name of a thread.

*setName()*: The setName() method sets the name of a thread.

**Creating Thread using Threading Module:**

To implement a new thread using the threading module, you have to do the following:

Define a new subclass of the Thread class.

Override the *\_\_init\_\_(self [,args])* method to add additional arguments.

Then override the *run(self [,args])* method to implement what the thread should do when started.

Once you have created the new Thread subclass, you can create an instance of it and then start a new thread by invoking the *start()* which will in turn call *run()* method. **(2)**

|  |  |
| --- | --- |
| **Starting a New Thread: (1)**  **EXAMPLE:**  *#!/usr/bin/python*  *import thread*  *import time*  *# Define a function for the thread*  *def print\_time( threadName, delay):*  *count = 0*  *while count < 5:*  *time.sleep(delay)*  *count += 1*  *print "%s: %s" % ( threadName, time.ctime(time.time()) )*  *# Create two threads as follows*  *try:*  *thread.start\_new\_thread( print\_time, ("Thread-1", 2, ) )*  *thread.start\_new\_thread( print\_time, ("Thread-2", 4, ) )*  *except:*  *print "Error: unable to start thread"*  *while 1:*  *pass*  **When the above code is executed, it produces following result:**  *Thread-1: Thu Jan 22 15:42:17 2009*  *Thread-1: Thu Jan 22 15:42:19 2009*  *Thread-2: Thu Jan 22 15:42:19 2009*  *Thread-1: Thu Jan 22 15:42:21 2009*  *Thread-2: Thu Jan 22 15:42:23 2009*  *Thread-1: Thu Jan 22 15:42:23 2009*  *Thread-1: Thu Jan 22 15:42:25 2009*  *Thread-2: Thu Jan 22 15:42:27 2009*  *Thread-2: Thu Jan 22 15:42:31 2009*  *Thread-2: Thu Jan 22 15:42:35 2009* | **Creating Thread using Threading Module: (2)**  **EXAMPLE:**  *#!/usr/bin/python*  *import threading*  *import time*  *exitFlag = 0*  *class myThread (threading.Thread):*  *def \_\_init\_\_(self, threadID, name, counter):*  *threading.Thread.\_\_init\_\_(self)*  *self.threadID = threadID*  *self.name = name*  *self.counter = counter*  *def run(self):*  *print "Starting " + self.name*  *print\_time(self.name, self.counter, 5)*  *print "Exiting " + self.name*  *def print\_time(threadName, delay, counter):*  *while counter:*  *if exitFlag:*  *thread.exit()*  *time.sleep(delay)*  *print "%s: %s" % (threadName, time.ctime(time.time()))*  *counter -= 1*  *# Create new threads*  *thread1 = myThread(1, "Thread-1", 1)*  *thread2 = myThread(2, "Thread-2", 2)*  *# Start new Threads*  *thread1.start()*  *thread2.start()*  *print "Exiting Main Thread"*  **When the above code is executed, it produces following result:**  *Starting Thread-1*  *Starting Thread-2*  *Exiting Main Thread*  *Thread-1: Thu Mar 21 09:10:03 2013*  *Thread-1: Thu Mar 21 09:10:04 2013*  *Thread-2: Thu Mar 21 09:10:04 2013*  *Thread-1: Thu Mar 21 09:10:05 2013*  *Thread-1: Thu Mar 21 09:10:06 2013*  *Thread-2: Thu Mar 21 09:10:06 2013*  *Thread-1: Thu Mar 21 09:10:07 2013*  *Exiting Thread-1*  *Thread-2: Thu Mar 21 09:10:08 2013*  *Thread-2: Thu Mar 21 09:10:10 2013*  *Thread-2: Thu Mar 21 09:10:12 2013*  *Exiting Thread-2* |

**Synchronizing Threads:**

The threading module provided with Python includes a simple-to-implement locking mechanism that will allow you to synchronize threads. A new lock is created by calling the Lock() method, which returns the new lock.

The *acquire(blocking)* method the new lock object would be used to force threads to run synchronously. The optional blocking parameter enables you to control whether the thread will wait to acquire the lock.

If blocking is set to 0, the thread will return immediately with a 0 value if the lock cannot be acquired and with a 1 if the lock was acquired. If blocking is set to 1, the thread will block and wait for the lock to be released.

The *release()* method of the the new lock object would be used to release the lock when it is no longer required.

|  |  |
| --- | --- |
| **EXAMPLE:**  *#!/usr/bin/python*  *import threading*  *import time*  *class myThread (threading.Thread):*  *def \_\_init\_\_(self, threadID, name, counter):*  *threading.Thread.\_\_init\_\_(self)*  *self.threadID = threadID*  *self.name = name*  *self.counter = counter*  *def run(self):*  *print "Starting " + self.name*  *# Get lock to synchronize threads*  *threadLock.acquire()*  *print\_time(self.name, self.counter, 3)*  *# Free lock to release next thread*  *threadLock.release()*  *def print\_time(threadName, delay, counter):*  *while counter:*  *time.sleep(delay)*  *print "%s: %s" % (threadName, time.ctime(time.time()))*  *counter -= 1*  *threadLock = threading.Lock()*  *threads = []*  *# Create new threads*  *thread1 = myThread(1, "Thread-1", 1)*  *thread2 = myThread(2, "Thread-2", 2)* | *# Start new Threads*  *thread1.start()*  *thread2.start()*  *# Add threads to thread list*  *threads.append(thread1)*  *threads.append(thread2)*  *# Wait for all threads to complete*  *for t in threads:*  *t.join()*  *print "Exiting Main Thread"*  **When the above code is executed, it produces following result:**  *Starting Thread-1*  *Starting Thread-2*  *Thread-1: Thu Mar 21 09:11:28 2013*  *Thread-1: Thu Mar 21 09:11:29 2013*  *Thread-1: Thu Mar 21 09:11:30 2013*  *Thread-2: Thu Mar 21 09:11:32 2013*  *Thread-2: Thu Mar 21 09:11:34 2013*  *Thread-2: Thu Mar 21 09:11:36 2013*  *Exiting Main Thread* |

**Multithreaded Priority Queue:**

The Queue module allows you to create a new queue object that can hold a specific number of items. There are following methods to control the Queue:

get(): The get() removes and returns an item from the queue.

put(): The put adds item to a queue.

qsize() : The qsize() returns the number of items that are currently in the queue.

empty(): The empty( ) returns True if queue is empty; otherwise, False.

full(): the full() returns True if queue is full; otherwise, False.