```
#SocketServer.py
# Edited to shorten for looking at TCP servers by John Zelle


"""Generic socket server classes.

This module tries to capture the various aspects of defining a server:

For socket-based servers:

- address family:
        - AF_INET{,6}: IP (Internet Protocol) sockets (default)
        - AF_UNIX: Unix domain sockets
        - others, e.g. AF_DECNET are conceivable (see <socket.h>
- socket type:
        - SOCK_STREAM (reliable stream, e.g. TCP)
        - SOCK_DGRAM (datagrams, e.g. UDP)

For request-based servers (including socket-based):

- client address verification before further looking at the request
        (This is actually a hook for any processing that needs to look
         at the request before anything else, e.g. logging)
- how to handle multiple requests:
        - synchronous (one request is handled at a time)
        - forking (each request is handled by a new process)
        - threading (each request is handled by a new thread)
"""

# Comments snipped here

__version__ = "0.4"


import socket
import sys
import os

# snipped some stuff

class BaseServer:

    """Base class for server classes.

    Methods for the caller:

    - __init__(server_address, RequestHandlerClass)
    - serve_forever()
    - handle_request()  # if you do not use serve_forever()
    - fileno() -> int   # for select()

    Methods that may be overridden:

    - server_bind()
    - server_activate()
    - get_request() -> request, client_address
    - verify_request(request, client_address)
    - server_close()
    - process_request(request, client_address)
    - close_request(request)
    - handle_error()

    Methods for derived classes:

    - finish_request(request, client_address)

    Class variables that may be overridden by derived classes or
    instances:

    - address_family
    - socket_type
    - reuse_address

    Instance variables:

    - RequestHandlerClass
    - socket

    """

    def __init__(self, server_address, RequestHandlerClass):
        """Constructor.  May be extended, do not override."""
        self.server_address = server_address
        self.RequestHandlerClass = RequestHandlerClass

    def server_activate(self):
        """Called by constructor to activate the server.

        May be overridden.

        """
        pass

    def serve_forever(self):
        """Handle one request at a time until doomsday."""
        while 1:
            self.handle_request()

    # The distinction between handling, getting, processing and
    # finishing a request is fairly arbitrary.  Remember:
    #
    # - handle_request() is the top-level call.  It calls
    #   get_request(), verify_request() and process_request()
    # - get_request() is different for stream or datagram sockets
    # - process_request() is the place that may fork a new process
    #   or create a new thread to finish the request
    # - finish_request() instantiates the request handler class;
    #   this constructor will handle the request all by itself

    def handle_request(self):
        """Handle one request, possibly blocking."""
        try:
            request, client_address = self.get_request()
        except socket.error:
            return
        if self.verify_request(request, client_address):
            try:
                self.process_request(request, client_address)
            except:
                self.handle_error(request, client_address)
                self.close_request(request)

    def verify_request(self, request, client_address):
        """Verify the request.  May be overridden.

        Return true if we should proceed with this request.

        """
        return 1

    def process_request(self, request, client_address):
        """Call finish_request.

        Overridden by ForkingMixIn and ThreadingMixIn.

        """
        self.finish_request(request, client_address)
        self.close_request(request)

    def server_close(self):
        """Called to clean-up the server.

        May be overridden.

        """
        pass

    def finish_request(self, request, client_address):
        """Finish one request by instantiating RequestHandlerClass."""
        self.RequestHandlerClass(request, client_address, self)

    def close_request(self, request):
        """Called to clean up an individual request."""
        pass

    def handle_error(self, request, client_address):
        """Handle an error gracefully.  May be overridden.

        The default is to print a traceback and continue.

        """
        print '-'*40
        print 'Exception happened during processing of request from',
        print client_address
        import traceback
        traceback.print_exc() # XXX But this goes to stderr!
        print '-'*40


class TCPServer(BaseServer):

    """Base class for various socket-based server classes.

    Defaults to synchronous IP stream (i.e., TCP).

    Methods for the caller:

    - __init__(server_address, RequestHandlerClass)
    - serve_forever()
    - handle_request()  # if you don't use serve_forever()
    - fileno() -> int   # for select()

    Methods that may be overridden:

    - server_bind()
    - server_activate()
    - get_request() -> request, client_address
    - verify_request(request, client_address)
    - process_request(request, client_address)
    - close_request(request)
    - handle_error()

    Methods for derived classes:

    - finish_request(request, client_address)

    Class variables that may be overridden by derived classes or
    instances:

    - address_family
    - socket_type
    - request_queue_size (only for stream sockets)
    - reuse_address

    Instance variables:

    - server_address
    - RequestHandlerClass
    - socket

    """

    address_family = socket.AF_INET

    socket_type = socket.SOCK_STREAM

    request_queue_size = 5

    allow_reuse_address = 0

    def __init__(self, server_address, RequestHandlerClass):
        """Constructor.  May be extended, do not override."""
        BaseServer.__init__(self, server_address, RequestHandlerClass)
        self.socket = socket.socket(self.address_family,
                                    self.socket_type)
        self.server_bind()
        self.server_activate()

    def server_bind(self):
        """Called by constructor to bind the socket.

        May be overridden.

        """
        if self.allow_reuse_address:
            self.socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        self.socket.bind(self.server_address)

    def server_activate(self):
        """Called by constructor to activate the server.

        May be overridden.

        """
        self.socket.listen(self.request_queue_size)

    def server_close(self):
        """Called to clean-up the server.

        May be overridden.

        """
        self.socket.close()

    def fileno(self):
        """Return socket file number.

        Interface required by select().

        """
        return self.socket.fileno()

    def get_request(self):
        """Get the request and client address from the socket.
```

```
        May be overridden.

        """
        return self.socket.accept()

    def close_request(self, request):
        """Called to clean up an individual request."""
        request.close()

#stuff snipped here (UDPServer, ForkingMixIn, Threading MixIn

class ForkingUDPServer(ForkingMixIn, UDPServer): pass
class ForkingTCPServer(ForkingMixIn, TCPServer): pass

class ThreadingUDPServer(ThreadingMixIn, UDPServer): pass
class ThreadingTCPServer(ThreadingMixIn, TCPServer): pass

# stuff snipped here (Unix sockets servers)

class BaseRequestHandler:

    """Base class for request handler classes.

    This class is instantiated for each request to be handled.  The
    constructor sets the instance variables request, client_address
    and server, and then calls the handle() method.  To implement a
    specific service, all you need to do is to derive a class which
    defines a handle() method.

    The handle() method can find the request as self.request, the
    client address as self.client_address, and the server (in case it
    needs access to per-server information) as self.server.  Since a
    separate instance is created for each request, the handle() method
    can define arbitrary other instance variariables.

    """

    def __init__(self, request, client_address, server):
        self.request = request
        self.client_address = client_address
        self.server = server
        try:
            self.setup()
            self.handle()
            self.finish()
        finally:
            sys.exc_traceback = None    # Help garbage collection

    def setup(self):
        pass

    def handle(self):
        pass

    def finish(self):
        pass


# The following two classes make it possible to use the same service
# class for stream or datagram servers.
# Each class sets up these instance variables:
# - rfile: a file object from which receives the request is read
# - wfile: a file object to which the reply is written
# When the handle() method returns, wfile is flushed properly

class StreamRequestHandler(BaseRequestHandler):

    """Define self.rfile and self.wfile for stream sockets."""

    # Default buffer sizes for rfile, wfile.
    # We default rfile to buffered because otherwise it could be
    # really slow for large data (a getc() call per byte); we make
    # wfile unbuffered because (a) often after a write() we want to
    # read and we need to flush the line; (b) big writes to unbuffered
    # files are typically optimized by stdio even when big reads
    # aren't.
    rbufsize = -1
    wbufsize = 0

    def setup(self):
        self.connection = self.request
        self.rfile = self.connection.makefile('rb', self.rbufsize)
        self.wfile = self.connection.makefile('wb', self.wbufsize)

    def finish(self):
        self.wfile.flush()
        self.wfile.close()
        self.rfile.close()

# stuff snipped here (DatagramRequestHandler)
```