How to scale MySQL Database Monitors by localizing them

The F5 MySQL database monitor works by forking a UNIX process that runs a MySQL database client that attempts to log into a database and perform a query. While this provides a very definitive answer as to the health of the database in question, it can also consume a lot of CPU on the F5 system. If a large amount of these monitors are required you can quickly get into a situation where the bulk of your CPU is spent health checking.

There are some options to minimize the problem:

1. You can set the timing of the health checks so that they run far less frequently. While this will allow more of them to exist, you now introduce a problem where an uncomfortable amount of time may pass before a health check is performed.
2. You can use a more simplistic health check like a TCP check. This will consume far less CPU, but the drawback is that you have only determined if something on the database is listening on that port. You don't know if it's a database, if it's a MySQL database, and if that database has the correct content or not.
3. Increase the capacity of the F5 device. Older devices that are single CPUs with double cores will exhaust resources quicker than newer and less expensive hardware that has more CPU resources to spend.

In most cases these solutions will resolve a configuration adequately, but in the case where several hundred databases are being monitored there may be a case where these actions are inadequate or don't scale sufficiently.

Moving the monitor

An alternative is to have the monitor run on the database server itself and then listen on a socket for a request for a health check. When the request arrives, the service can check the database and then report back the health of the system. This would allow you to use a TCP health check on the F5, which are very fast and use very little resources, and amortize the resources required to check the databases across the database servers themselves. There are some additional advantages and architectures that can be explored as well, but for now I'll demonstrate a simple example of this.

For the purposes of this example, the MySQL database is assumed to be running on flavor of Linux. While it is possible for MySQL to run on windows it is predominantly found on Linux based systems and thus certain problems were solved in a 'Linux manner'. To use this solution on Windows, the daemon code would need to be modified since it is currently using Linux mechanisms to work.

Creating a test environment

MySQL server can be installed easily on a number of UNIX systems. For example:

```
Ubuntu:
sudo apt-get install mysql-server

Centos:
sudo yum install mysql-server
/etc/init.d/mysqld start
```

Once you have MySQL installed on the test server, you can access MySQL locally using the mysql client tool:

```
mysql –u root –p
```

You can quickly see what databases are on the server by typing:

```
SHOW DATABASES;
```

Which will result in a table being returned that looks something like this:

```
mysql> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| test               |
+--------------------+
4 rows in set (0.01 sec)
```

Creating a database is also very simple, as is deleting a database:

```
CREATE DATABASE database name;
DROP DATABASE database name;
```

To use a database that is already created, log in and open the database with the use command:

```
USE events;
```

You can now look at the details of this specific database. For instance you can see the list of tables in this database:

```
SHOW tables;
```

To create a simple table that just has a name and a key, you would enter the following:

```
CREATE TABLE potluck (id INT NOT NULL PRIMARY KEY
AUTO_INCREMENT, name VARCHAR(20));
```

Again, you can use show <table name>; to verify that the table was created. If you want to look at the details of the table you would use DESCRIBE:

```
mysql>DESCRIBE potluck;
+-------------+-------------+------+-----+---------+----------------+
| Field       | Type        | Null | Key | Default | Extra          |
+-------------+-------------+------+-----+---------+----------------+
| id          | int(11)     | NO   | PRI | NULL    | auto_increment |
| name        | varchar(20) | YES  |     | NULL    |                |
+-------------+-------------+------+-----+---------+----------------+
2 rows in set (0.01 sec)
```

To insert data into the table you use the INSERT command as follows:

```
INSERT INTO `potluck` (`id`,`name’) VALUES (NULL, "John");
Query OK, 1 row affected (0.00 sec)
```

To view the contents of the table, we can write a simple database query:

```
mysql> SELECT * FROM potluck;
+----+-------+
| id | name  |
+----+-------+
|  1 | John  |
+----+-------+
1 rows in set (0.00 sec)
```

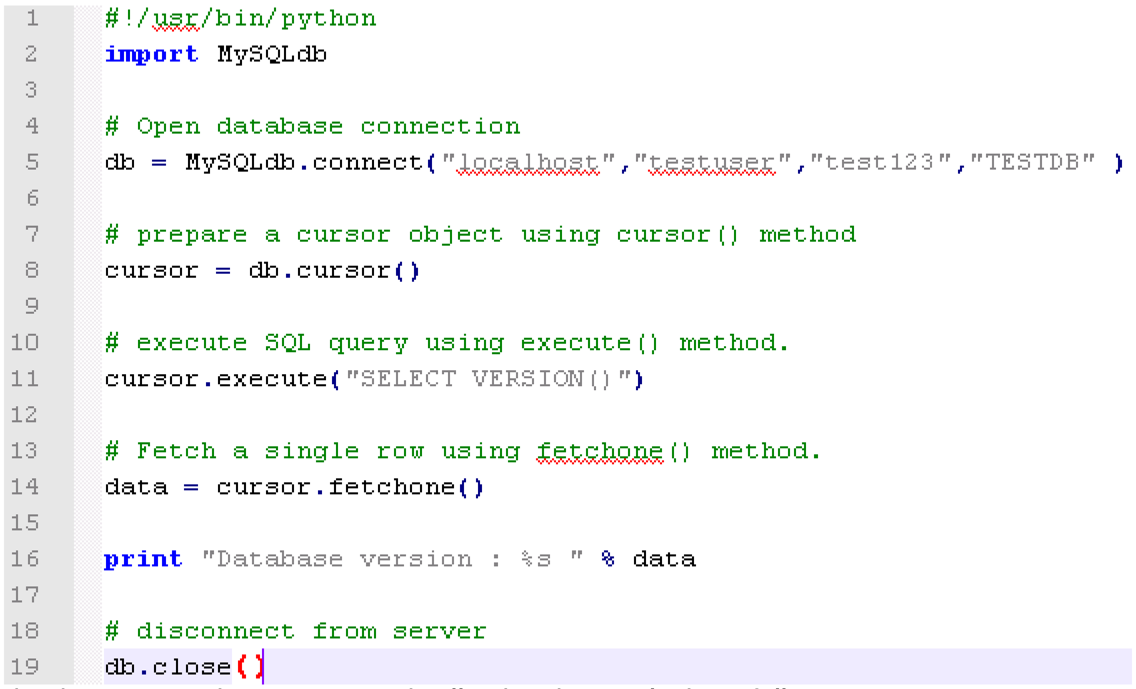
With this information, you should be able to install and setup a MySQL database for testing. Its advised that you read through the rest of this document first to see how the code is configured so that the test environment mimics what the code is assuming.

Breaking the project down

There are three distinct pieces to this daemon: a module that queries the database, a module that provides a simple TCP server and some code that allows us to daemonize the project. Breaking down the pieces and solving them one at a time will be easier to explain and allow easier testing of the logic. Once the pieces are in place, they can be combined and then tested for completeness.

Writing a database query

The easiest part of this project is dealing with the database query. You will need to ensure that the MySQLdb python module is installed before continuing. A simple example of connection to the



```
1   #!/usr/bin/python
2   import MySQLdb
3
4   # Open database connection
5   db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )
6
7   # prepare a cursor object using cursor() method
8   cursor = db.cursor()
9
10  # execute SQL query using execute() method.
11  cursor.execute("SELECT VERSION()")
12
13  # Fetch a single row using fetchone() method.
14  data = cursor.fetchone()
15
16  print "Database version : %s " % data
17
18  # disconnect from server
19  db.close()
```

database, executing a query, and collecting the results is as follows:

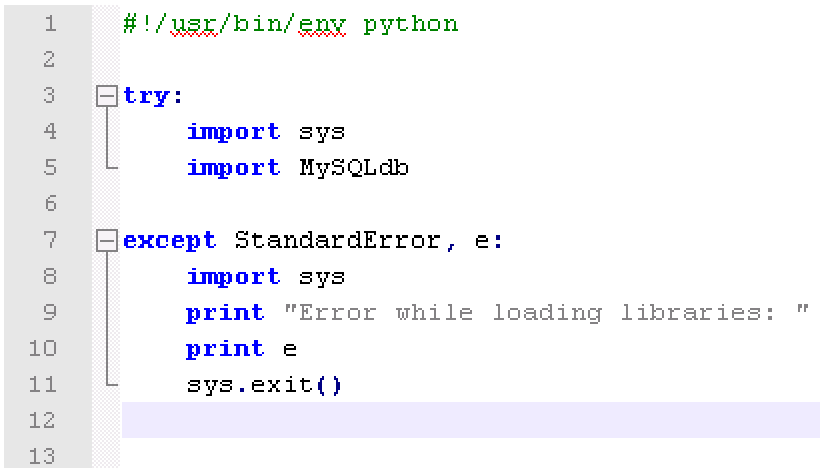
Obviously, the start of this module will need to import MySQLdb. The first step is to make a connection to the database which is done on line 5. Here, four arguments are passed to connect: the server's address, the username, password, and the name of the database. If the connection information is correct and the database is up, db will be an object that allows us to manipulate the database TESTDB on server localhost.

Next we obtain a cursor object, so that we can execute queries on it. We get the cursor from the db object on line 8. Using the cursor object, we can then use its execute() method, in this case getting the version of the database as seen on line 11. To see the results of our query, we use the fetchone() method of the cursor object. This retrieves the next row of a query result set and returns a single sequence.

We print the data collected from the fetchone() method and then lastly close our connection to the database using the close() method on line 19.

We will use this basic framework for our database check since it establishes that the database server is up and listening, the database is up and valid, a known table in that database is accessible, and known data pulled from that table matches. If any of this should fail, we will consider the health check as failed. You can implement more granular feedback that would describe what specifically fails but that may complicate the code unnecessarily.

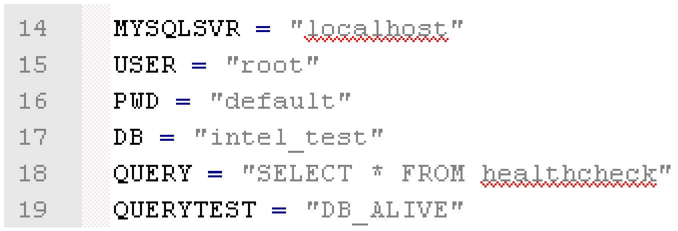
To start out our dbCheck module, we will import the required modules and add an interpreter line at the top of the file so it can be executed directly:



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

try:
    import sys
    import MySQLdb

except StandardError, e:
    import sys
    print "Error while loading libraries: "
    print e
    sys.exit()

```

It's not necessary to encapsulate the imports within a try:except clause, but it makes errors more informative if there is a problem with the import.

Next, in order to make it easier to modify the configuration, we extract them out and put them at the top of the file. This way if a query needs to be changed or credentials modified they are easy to find.

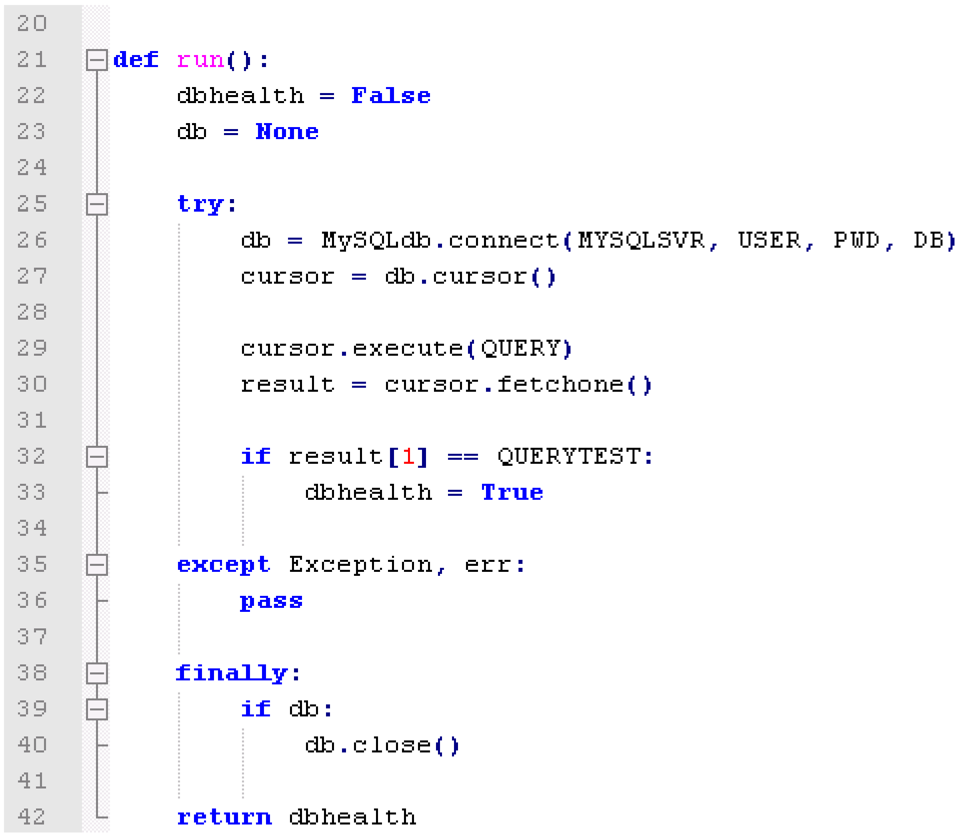


```
MYSQLSVR = "localhost"
USER = "root"
PWD = "default"
DB = "intel_test"
QUERY = "SELECT * FROM healthcheck"
QUERYTEST = "DB_ALIVE"
```

Since this routine is simple, we will keep it to one function called run(). The start of run() sets dbHealth to False. Within the try block later, we will have to pass all of the code described previously before dbHealth is set to True. It's a small point, but this makes the logic simpler. We also set db to None so if the connect call fails we know the call to get a cursor will result in an exception.

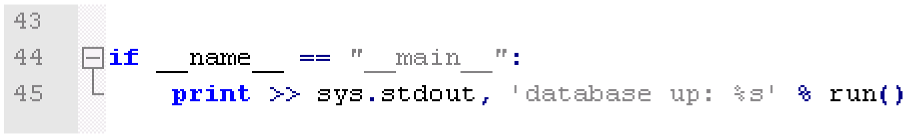
Entering the try block, we run through the same code described previously. The result will be a tuple consisting of an id and name from the row we collected. The id isn't important to us but the name is. We compare this against QUERYTEST and if it matches set dbHealth to True. If any of these operations fail, an exception gets raised or we drop through to the finally block and dbHealth is still False.

The finally block checks to see if db isn't None and if it's not, closes the database connection. Lastly, we return the result of dbHealth.



```
def run():
    dbhealth = False
    db = None

    try:
        db = MySQLdb.connect(MYSQLSVR, USER, PWD, DB)
        cursor = db.cursor()

        cursor.execute(QUERY)
        result = cursor.fetchone()

        if result[1] == QUERYTEST:
            dbhealth = True

    except Exception, err:
        pass

    finally:
        if db:
            db.close()

    return dbhealth
```

The last part of the file just provides a means to run the code and return the result:



```
if __name__ == "__main__":
    print >> sys.stdout, 'database up: %s' % run()
```

That wraps up the database check. At this point you would want to make sure that this returns True. If not, determine what is missing or not configured correctly. You should have a database that will reflect the values that are being queried for here. Once you can get a positive response you can also misconfigure values to check failures.

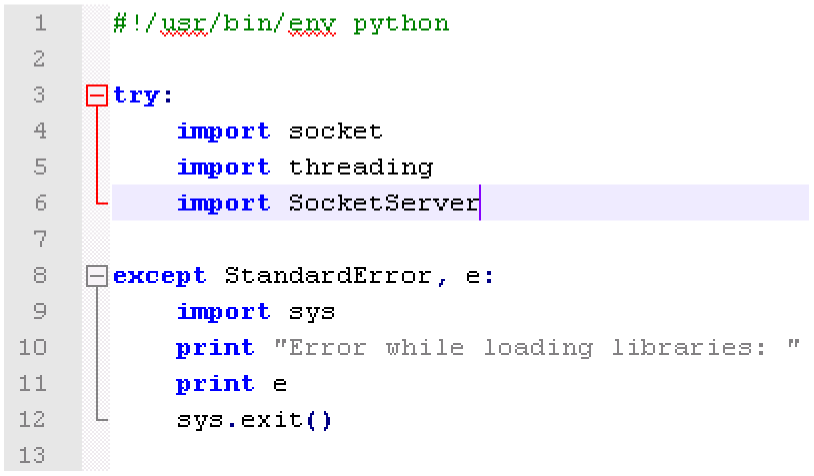
Writing a simple TCP server

There are a couple of options available to us for writing a TCP server. A very good python library is Twisted which provides support for web servers, publish/subscribe servers, mail clients, SSL, and more. It's a very well documented library with a lot of examples but it does require us to install another library and could be arguably more robust than we need.

We could also just use raw sockets and handle the messages ourselves. While this is tempting given the requirements we have, it might be more trouble than it's worth to get everything right.

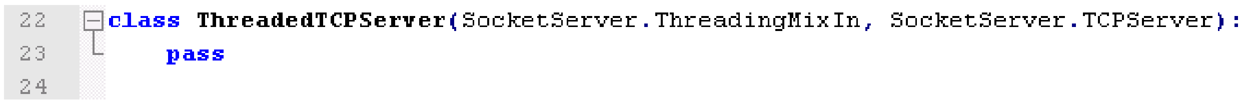
Luckily, there is a nice built-in library in python specifically for building network servers. It defines classes for handling synchronous network requests over TCP, UDP, Unix streams, and Unix datagrams. It also provides mix-in classes for easily converting servers to use a separate thread or process for each request, depending on what is most appropriate for your situation. It's lightweight and is perfect middle ground for what we need.

To start with, we add the interpreter line again and import our needed libraries. The socket import is for the client we test with, covered below.



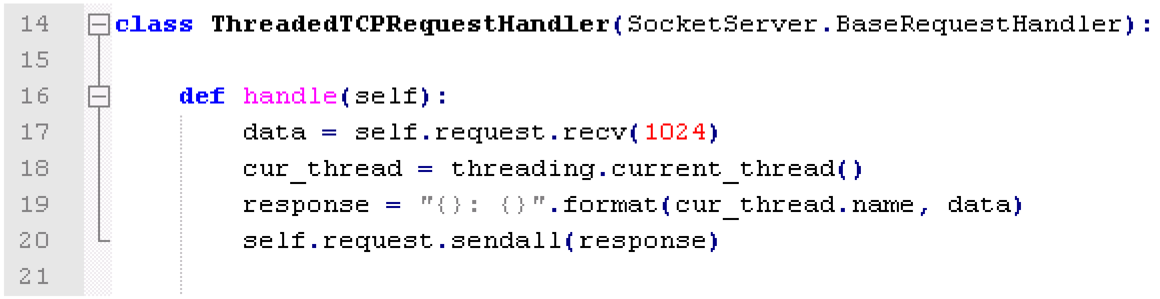
```
#!/usr/bin/env python

try:
    import socket
    import threading
    import SocketServer

except StandardError, e:
    import sys
    print "Error while loading libraries: "
    print e
    sys.exit()

```

Next, we create a ThreadedTCPServer. This is as easy as inheriting from a Threading mix in and a TCPServer. There are other methods that can be overwritten but that is for complex applications and unnecessary for what we are doing.



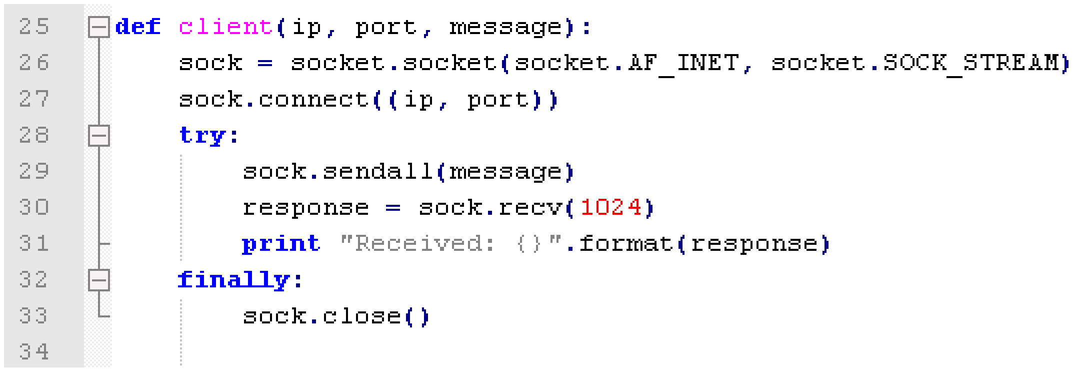
```
class ThreadedTCPServer(SocketServer.ThreadingMixIn, SocketServer.TCPServer):
    pass

```

To customize what we do when we get a client connection we write a Request Handler. This gets called by the ThreadedTCPServer any time a client connection is made and allows us to customize what we want the TCP server to do. In this case, we will just get the thread name and then echo back the client's message along with the name of the thread that is processing the response.



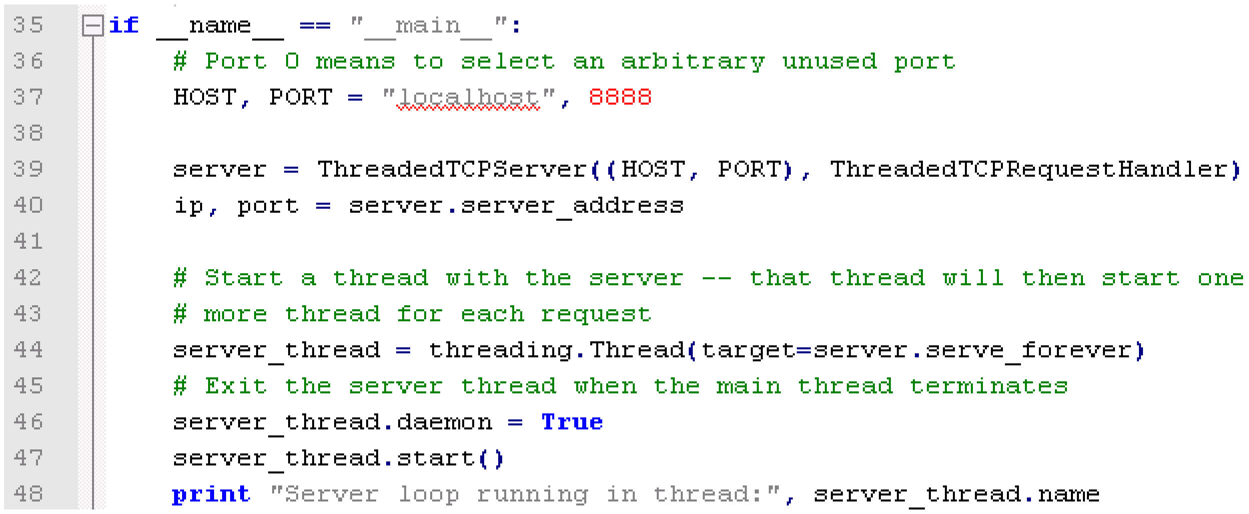
```
class ThreadedTCPRequestHandler(SocketServer.BaseRequestHandler):

    def handle(self):
        data = self.request.recv(1024)
        cur_thread = threading.current_thread()
        response = "{}: {}".format(cur_thread.name, data)
        self.request.sendall(response)

```

At this point, we segue to writing a simple client that sends a message to a TCP server. We will use this later when testing the TCP server. The client is very simple, we create a socket on line 26 and then connect to the IP and port that is passed to the function. Then we send the message on that port and receive 1K of data back from it, printing the response. Lastly, we close the socket.



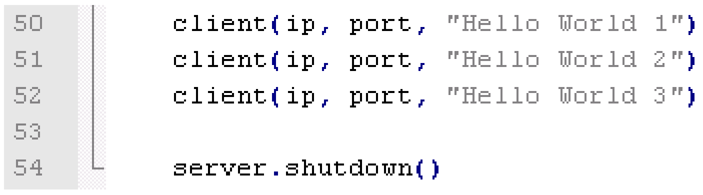
```
25  def client(ip, port, message):
26      sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
27      sock.connect((ip, port))
28      try:
29          sock.sendall(message)
30          response = sock.recv(1024)
31          print "Received: {}".format(response)
32      finally:
33          sock.close()
34
```

The remainder of the code starts the server up and then creates a few client connections to test that the server is up, listening, and processing requests. We start off defining the host and port that the server will listen on, in this case localhost:8888. We create the server object using our ThreadedTCPServer class defined above and pass it the host, port and the Request Handler class. This creates the server object and initializes its internal state. Next, we create a server thread that will execute the method in server.serve_forever on line 44. Setting the server thread daemon to True means it will exit when the main thread stops and then on line 47 we start the thread.



```
35  if __name__ == "__main__":
36      # Port 0 means to select an arbitrary unused port
37      HOST, PORT = "localhost", 8888
38
39      server = ThreadedTCPServer((HOST, PORT), ThreadedTCPRequestHandler)
40      ip, port = server.server_address
41
42      # Start a thread with the server -- that thread will then start one
43      # more thread for each request
44      server_thread = threading.Thread(target=server.serve_forever)
45      # Exit the server thread when the main thread terminates
46      server_thread.daemon = True
47      server_thread.start()
48      print "Server loop running in thread:", server_thread.name
```

The last part of this code runs the client function sending a few messages and then we finish up by shutting down the TCP Server.



```
50      client(ip, port, "Hello World 1")
51      client(ip, port, "Hello World 2")
52      client(ip, port, "Hello World 3")
53
54      server.shutdown()
```

You should receive a response similar to below which indicates that the server is running on Thread-1 and the respective threads handled the three client requests.

```
f5@xubuntu-vm:~/python/archive$ ./server6.py
Server loop running in thread: Thread-1
Received: Thread-2: Hello World 1
Received: Thread-3: Hello World 2
Received: Thread-4: Hello World 3
f5@xubuntu-vm:~/python/archive$
```

That wraps up the second of our modules. The final step is to figure out how we wrap this all into a daemon so that it runs in the background on the server.

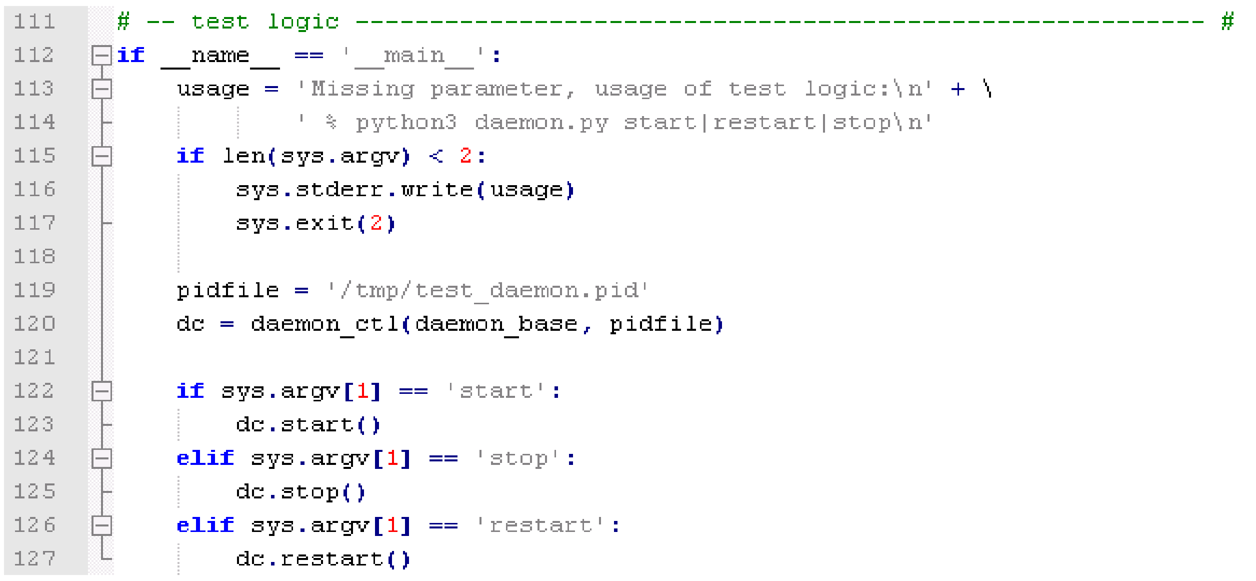
Writing a daemon wrapper

Creating a daemon out of our process is a bit involved, but fortunately there are some pretty good examples already available. Python doesn't have any built in modules for doing this and the best source of information is the book, "Advanced Programming in the UNIX Environment" by Stevens. In there he describes the double-fork technique which essentially splits out process while killing off the original process and thus daemonizing our process. Some others have taken this technique and wrapped it into a class, which can be found here:

http://www.jejik.com/articles/2007/02/a_simple_unix_linux_daemon_in_python/

We are going to take a slightly different approach by separating out the daemon code from the controlling code which was contributed in the comments section further in the article.

On this module, we are going to go backwards and work from the entry point to the lower level parts of the code. The test logic is very straightforward and is what we will use later for starting the daemon.

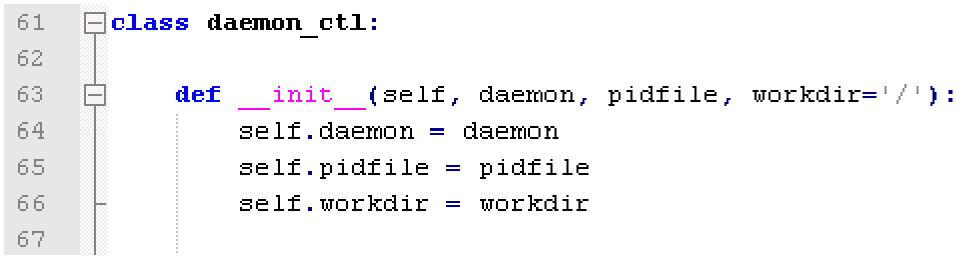


```python
# -- test logic ---------------------------------------------------------- #
if __name__ == '__main__':
    usage = 'Missing parameter, usage of test logic:\n' + \
            ' % python3 daemon.py start|restart|stop\n'
    if len(sys.argv) < 2:
        sys.stderr.write(usage)
        sys.exit(2)

    pidfile = '/tmp/test_daemon.pid'
    dc = daemon_ctl(daemon_base, pidfile)

    if sys.argv[1] == 'start':
        dc.start()
    elif sys.argv[1] == 'stop':
        dc.stop()
    elif sys.argv[1] == 'restart':
        dc.restart()
```

The entry point starts off by checking to see if the script was called with an argument. If it is not, then it prints out the usage message and errors out. Considering that we do have a command and an argument, we establish a location and name for our PID file, and then create a daemon_ctl object. When we create the daemon_ctl object we pass it a daemon class, in this case the daemon_base class, and a pidfile location.

Based on the argument passed, we call the daemon_ctl object method start(), stop(), or restart(). This controls what happens with the daemon that is created based on the class we passed to daemon_ctl.

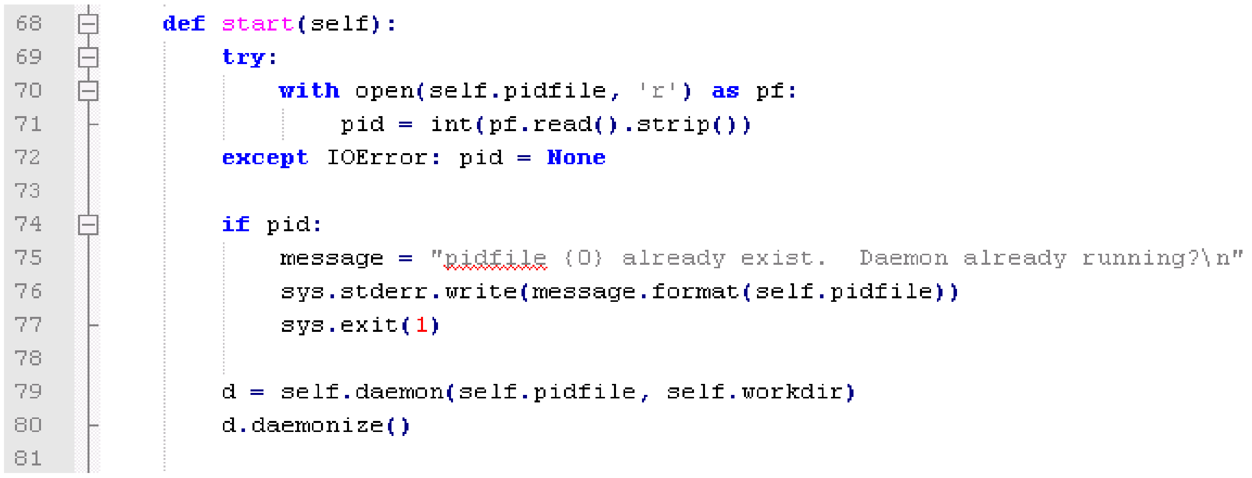
The next class is the daemon_ctl class. This class controls starting and stopping the daemon. The init method simply takes the classes and objects we pass in and saves them to instance variables.



```
61  class daemon_ctl:
62
63      def __init__(self, daemon, pidfile, workdir='/'):
64          self.daemon = daemon
65          self.pidfile = pidfile
66          self.workdir = workdir
67
```

The start method attempts to open the pidfile on line 70 and if it finds it strips out the process ID from the pidfile. Otherwise it raises an exception which goes to line 72 where the process ID is set to None. If there is a process ID, then it's assumed that the daemon is already running and an error message is written out on line 76 and the process exits on line 77.

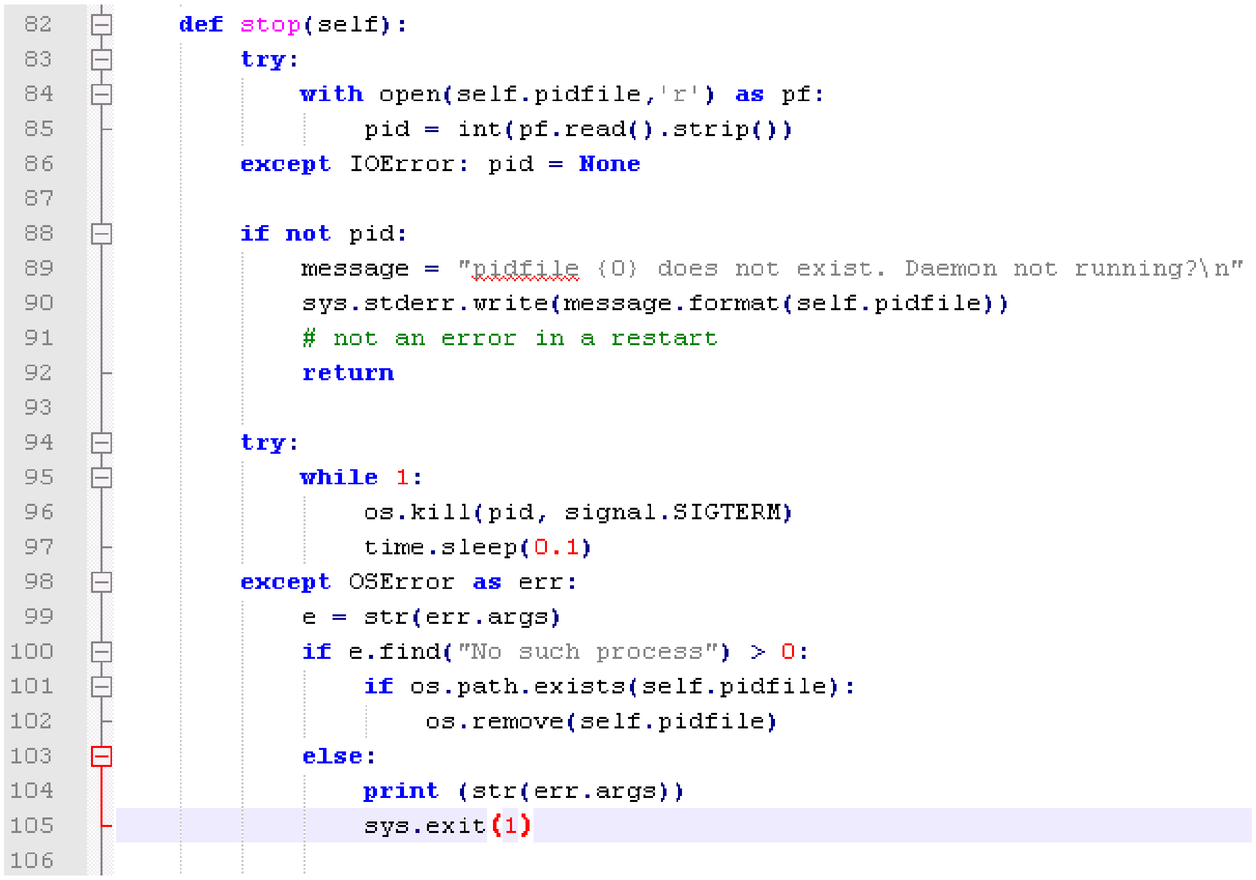
If the PID check shows the daemon is not running then we create a daemon process by creating an object based on the daemon class passed to daemon_ctl. We pass in the pidfile and working directory to the daemon object. Lastly we call the daemonize method in the daemon class to make the process a daemon.



```
68      def start(self):
69          try:
70              with open(self.pidfile, 'r') as pf:
71                  pid = int(pf.read().strip())
72          except IOError: pid = None
73
74          if pid:
75              message = "pidfile {0} already exist.  Daemon already running?\n"
76              sys.stderr.write(message.format(self.pidfile))
77              sys.exit(1)
78
79          d = self.daemon(self.pidfile, self.workdir)
80          d.daemonize()
81
```

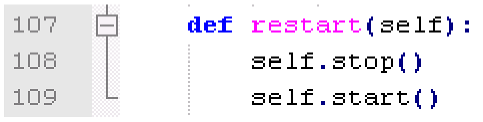
The stop method attempts to open the pidfile on line 84 and if it finds it strips out the process ID from the pidfile on line 85. Otherwise it raises an exception which goes to line 86 where the process ID is set to None. If there is no process ID, then it's assumed that the process is not running and an error message is written out on line 90 and the method returns. If there is a PID then we attempt to kill it by sending a SIGTERM signal to the process ID on line 96. We wait and then repeat indefinitely until the process dies.

Once the process is exiting the os.kill method will raise an exception, and the program flows to line 99. We check the error message and if it contains "No such process" then we know the exception was raised because the process no longer exists. At that point, we remove the pidfile if it exists on line 102 and then return to the caller. If the exception was raised for another reason, then we print the error and the exit the process.



```
 82     def stop(self):
 83         try:
 84             with open(self.pidfile,'r') as pf:
 85                 pid = int(pf.read().strip())
 86         except IOError: pid = None
 87 
 88         if not pid:
 89             message = "pidfile {0} does not exist. Daemon not running?\n"
 90             sys.stderr.write(message.format(self.pidfile))
 91             # not an error in a restart
 92             return
 93 
 94         try:
 95             while 1:
 96                 os.kill(pid, signal.SIGTERM)
 97                 time.sleep(0.1)
 98         except OSError as err:
 99             e = str(err.args)
100             if e.find("No such process") > 0:
101                 if os.path.exists(self.pidfile):
102                     os.remove(self.pidfile)
103             else:
104                 print (str(err.args))
105                 sys.exit(1)
106 
```

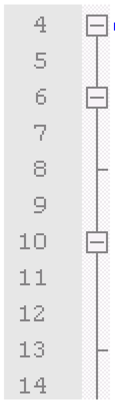
The last method is the restart method. This method simply calls the stop method and then the start method, so nothing special.



```
107     def restart(self):
108         self.stop()
109         self.start()
```

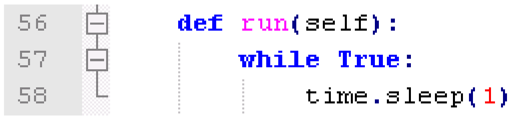
The final class is the daemon_base class. There are two key methods in the class, the daemonize method and the run method. The daemonize method employs the double fork method mentioned earlier to daemonize the process. The run method is what the daemon ‘does’ when it runs. Ideally, you would inherit the daemon_base class and override the run method to perform whatever task you want. We will do this later when we start to combine the modules.

Getting into the daemon_base class:



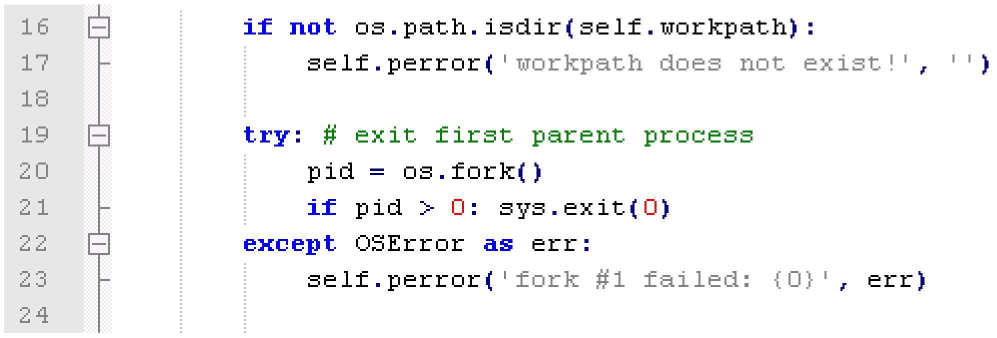
```python
class daemon_base:

    def __init__(self, pidfile, workpath='/'):
        self.pidfile = pidfile
        self.workpath = workpath

    def perror(self, msg, err):
        msg = msg + '\n'
        sys.stderr.write(msg.format(err))
        sys.exit(1)

```

The init method saves the passed in object to instance variables and the perror method simply provides a means to print an error and then exit the process. The default run method right now just enters an infinite while loop that sleeps for one second every loop.



```python
    def run(self):
        while True:
            time.sleep(1)
```

The last method to cover is the daemonize method. The first step is to check and ensure that the work path is valid on line 16. If it is not perror is called that prints the error and exits the process. The next step is to perform the first fork on line 20. If the fork is successful, then the fork method will return a PID in which case we kill this process. If this doesn't work an exception is raised that calls perror that prints the error message and exits.



```python
16          if not os.path.isdir(self.workpath):
17              self.perror('workpath does not exist!', '')
18
19          try: # exit first parent process
20              pid = os.fork()
21              if pid > 0: sys.exit(0)
22          except OSError as err:
23              self.perror('fork #1 failed: {0}', err)
24
```

If the first fork is a success then the next step is to decouple from the parent environment. First we attempt to change the directory to the work path directory. If that fails, we call the perror method. Next, we call setsid that creates a session and sets the process group ID. Next we call umask to set the default file permission for new files. Once this is complete, we attempt the second fork. This is performed in the same manner as the previous fork.

```
        # decouple from parent environment
        try: os.chdir(self.workpath)
        except OSError as err:
            self.perror('path change failed: {0}', err)

        os.setsid()
        os.umask(0)

        try: # exit from second parent
            pid = os.fork()
            if pid > 0: sys.exit(0)
        except OSError as err:
            self.perror('fork #2 failed: {0}', err)

```

If the second fork succeeds, we redirect all the standard file descriptors and then write the process ID into the pidfile.

```
        # redirect standard file descriptors
        sys.stdout.flush()
        sys.stderr.flush()
        si = open(os.devnull, 'r')
        so = open(os.devnull, 'a+')
        se = open(os.devnull, 'a+')
        os.dup2(si.fileno(), sys.stdin.fileno())
        os.dup2(so.fileno(), sys.stdout.fileno())
        os.dup2(se.fileno(), sys.stderr.fileno())

        # write pidfile
        atexit.register(os.remove, self.pidfile)
        pid = str(os.getpid())
        with open(self.pidfile,'w+') as f:
            f.write(pid + '\n')
```

The final step is to call the run method which begins the functional part of the daemon.

```
        self.run()

```

At this point you can test if the daemon code works. Albeit the daemon doesn't really do much but loop forever but you can get a sense that you can start the process, stop the process and restart the process. The next section, we will start combining these modules into one program.

Combining the project

The most logical place to start combining is the database check and the TCP server. The file starts off with the usual interpreter line and some imports. We are also going to make some modifications by adding python logging to the project.

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

import logging
import sys
import SocketServer
import MySQLdb

logging.basicConfig(level=logging.DEBUG, format='%(name)s: %(message)s',)

```

The TCP Server is setup the same way by inheriting from the Threading mixin and TCP Server.

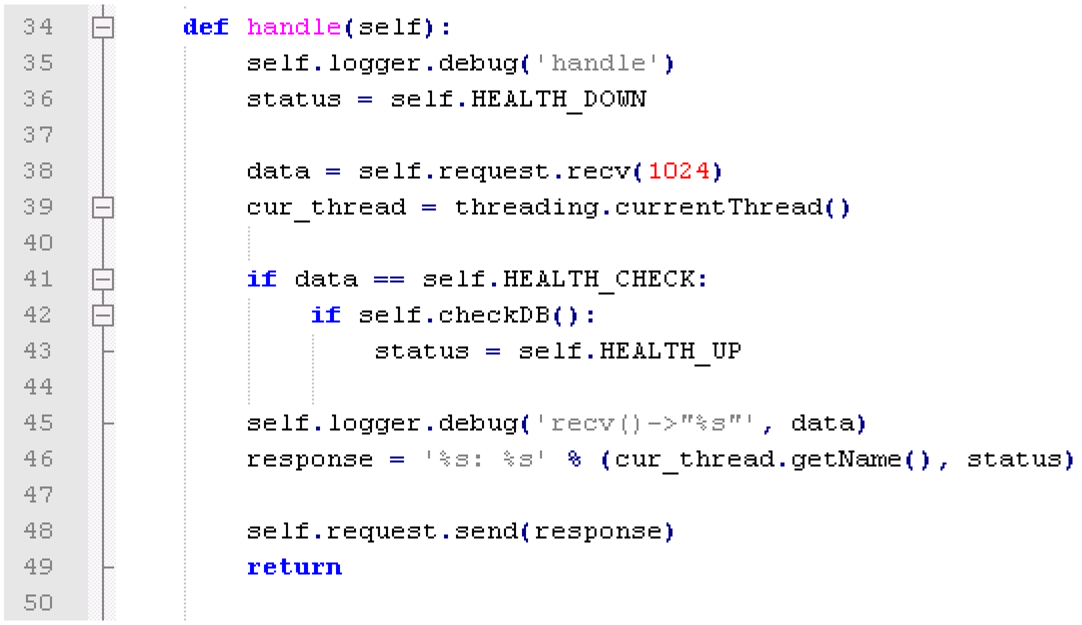
```python
class EchoServer(SocketServer.ThreadingMixIn, SocketServer.TCPServer):
    pass
```

The MySQL database check will get integrated into the request handler class of the TCP Server. The Request handler starts off much like it did previously:

```python
class EchoRequestHandler(SocketServer.BaseRequestHandler):

    def __init__(self, request, client_address, server):
        self.logger = logging.getLogger('EchoRequestHandler')
        self.logger.debug('__init__')

        self.HEALTH_UP = "SERVER_UP"
        self.HEALTH_DOWN = "SERVER_DN"
        self.HEALTH_CHECK = "8675309"

        self.MYSQLSVR = "localhost"
        self.USER = "root"
        self.PWD = "default"
        self.DB = "intel_test"
        self.QUERY = "SELECT * FROM healthcheck"
        self.QUERYTEST = "DB_ALIVE"

        SocketServer.BaseRequestHandler.__init__(self, request, client_address, server)
        return

    def setup(self):
        self.logger.debug('setup')
        return SocketServer.BaseRequestHandler.setup(self)

```

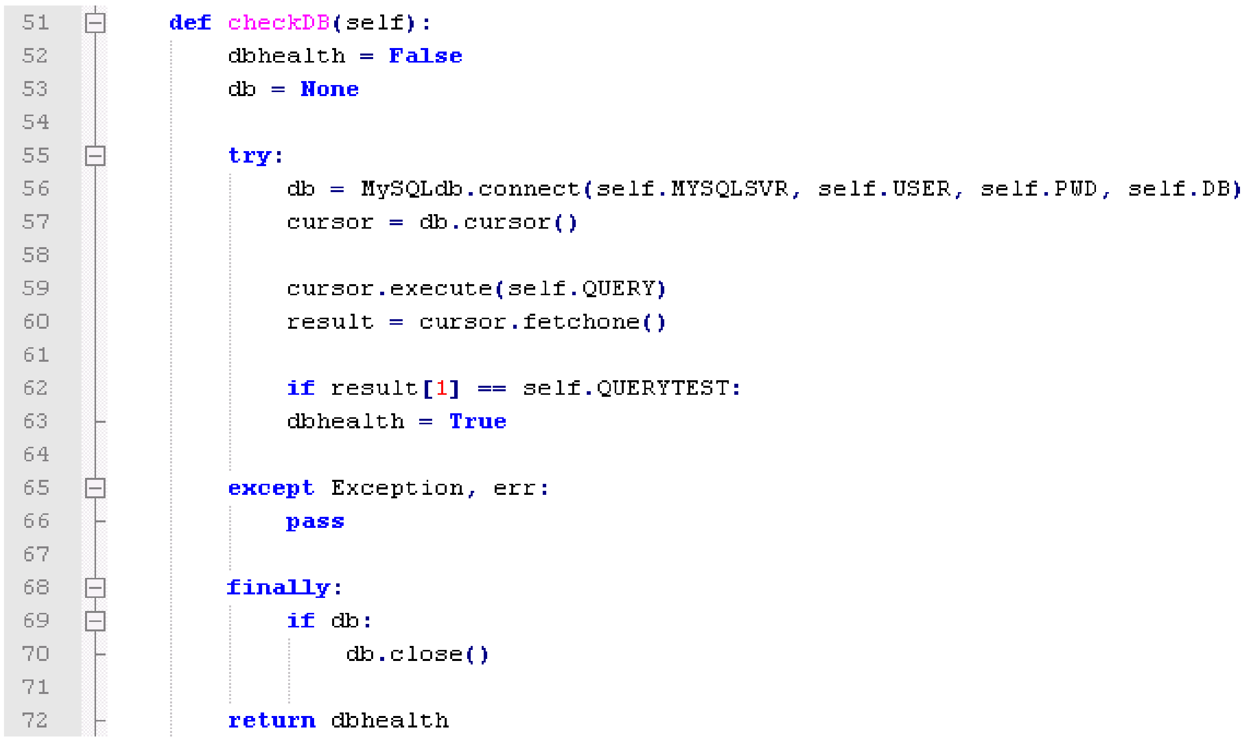
The handle method is modified slightly to assume a status of HEALTH_DOWN in the beginning. We receive the data from the client on line 38 and check it against the HEALTH_CHECK signature on line 41. This is a very simplistic mechanism to ensure that only a proper request makes a query against the database. There are much better cryptographic mechanisms that you could use in lieu of this method and you should consider one of them if you are really concerned about unauthorized health checks sapping database resources.

Next, we run the checkDB method which returns true if the DB check succeeds and False if it does not. If the check is good, we change the status to HEALTH_UP on line 43. Next we build a response on line 46 and then send that back to the client and then exit the handler function.



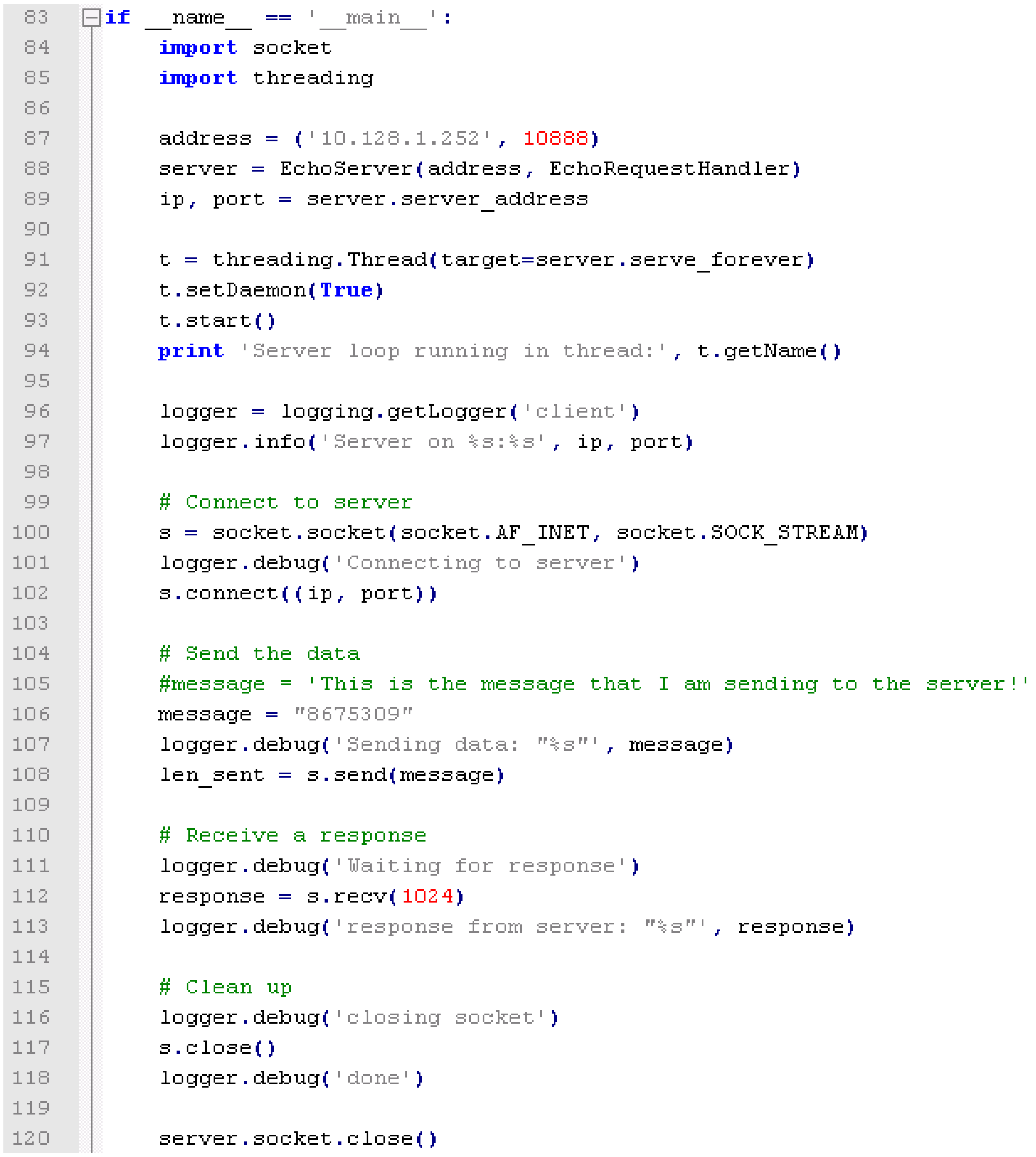
```
    def handle(self):
        self.logger.debug('handle')
        status = self.HEALTH_DOWN

        data = self.request.recv(1024)
        cur_thread = threading.currentThread()

        if data == self.HEALTH_CHECK:
            if self.checkDB():
                status = self.HEALTH_UP

        self.logger.debug('recv()->"%s"', data)
        response = '%s: %s' % (cur_thread.getName(), status)

        self.request.send(response)
        return

```

The checkDB method is the same as what we did earlier and merges easily with the rest of the code:



```
    def checkDB(self):
        dbhealth = False
        db = None

        try:
            db = MySQLdb.connect(self.MYSQLSVR, self.USER, self.PWD, self.DB)
            cursor = db.cursor()

            cursor.execute(self.QUERY)
            result = cursor.fetchone()

            if result[1] == self.QUERYTEST:
            dbhealth = True

        except Exception, err:
            pass

        finally:
            if db:
                db.close()

        return dbhealth
```

The final thing to do is create an entry point that creates the TCP server and then creates a TCP client that connects to the server and sends a HEALTH_CHECK request and prints the reply.



```python
if __name__ == '__main__':
    import socket
    import threading

    address = ('10.128.1.252', 10888)
    server = EchoServer(address, EchoRequestHandler)
    ip, port = server.server_address

    t = threading.Thread(target=server.serve_forever)
    t.setDaemon(True)
    t.start()
    print 'Server loop running in thread:', t.getName()

    logger = logging.getLogger('client')
    logger.info('Server on %s:%s', ip, port)

    # Connect to server
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    logger.debug('Connecting to server')
    s.connect((ip, port))

    # Send the data
    #message = 'This is the message that I am sending to the server!'
    message = "8675309"
    logger.debug('Sending data: "%s"', message)
    len_sent = s.send(message)

    # Receive a response
    logger.debug('Waiting for response')
    response = s.recv(1024)
    logger.debug('response from server: "%s"', response)

    # Clean up
    logger.debug('closing socket')
    s.close()
    logger.debug('done')

    server.socket.close()
```

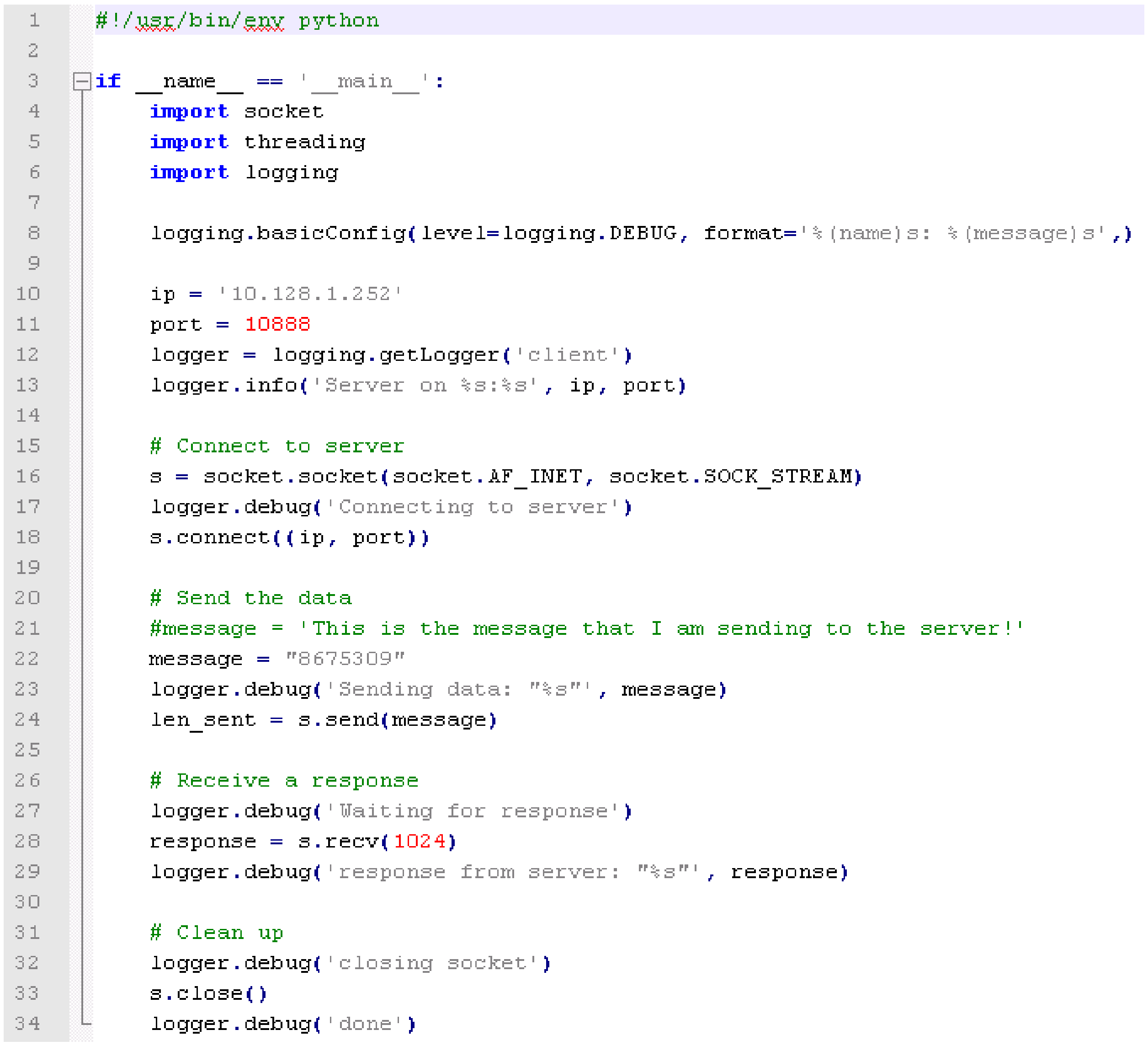
The output should look something like the following:

```
f5@xubuntu-vm:~/python/archive$ ./server8.py
Server loop running in thread: Thread-1
client: Server on 10.128.1.252:10888
client: Connecting to server
client: Sending data: "8675309"
client: Waiting for response
```

```
EchoRequestHandler: __init__
EchoRequestHandler: setup
EchoRequestHandler: handle
EchoRequestHandler: recv()->"8675309"
client: response from server: "Thread-2: SERVER_UP"
client: closing socket
client: done
EchoRequestHandler: finish
f5@xubuntu-vm:~/python/archive$
```

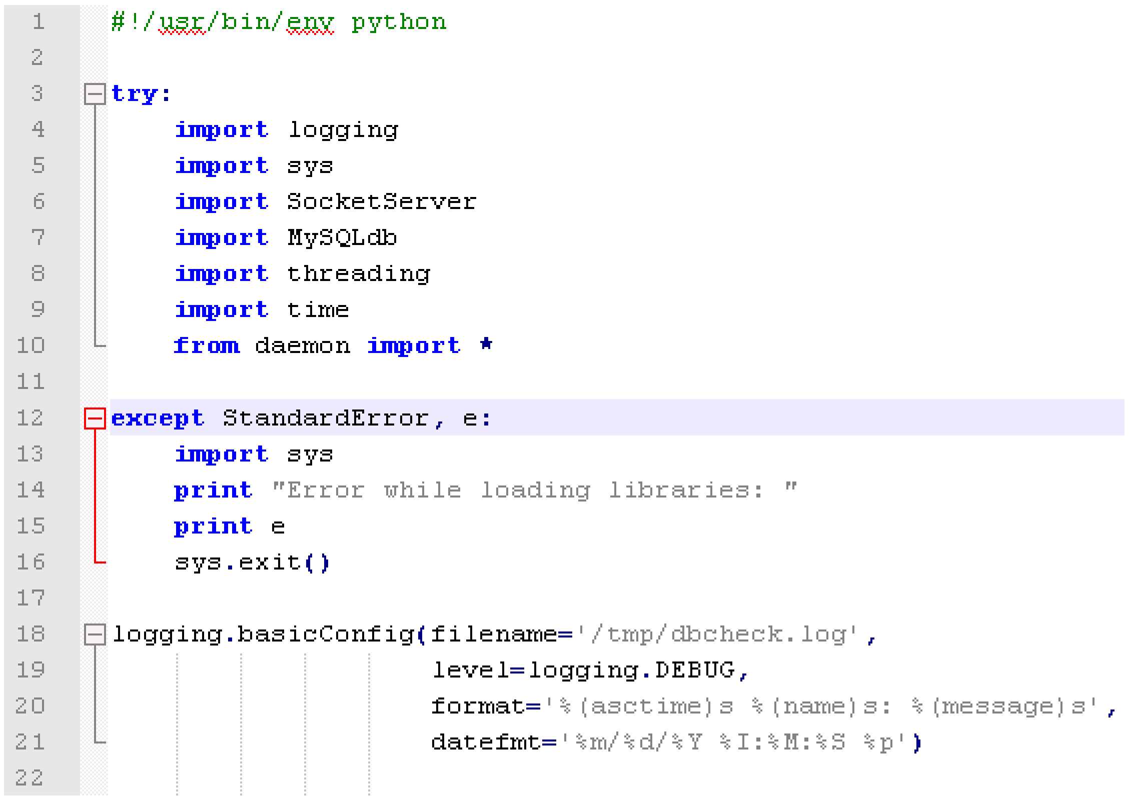
Final steps

Before we finish combining the rest of the code, we will build a client that we can use to test that the server works and is providing proper responses. Most of this client is derived from the previous example:



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

if __name__ == '__main__':
    import socket
    import threading
    import logging

    logging.basicConfig(level=logging.DEBUG, format='%(name)s: %(message)s',)

    ip = '10.128.1.252'
    port = 10888
    logger = logging.getLogger('client')
    logger.info('Server on %s:%s', ip, port)

    # Connect to server
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    logger.debug('Connecting to server')
    s.connect((ip, port))

    # Send the data
    #message = 'This is the message that I am sending to the server!'
    message = "8675309"
    logger.debug('Sending data: "%s"', message)
    len_sent = s.send(message)

    # Receive a response
    logger.debug('Waiting for response')
    response = s.recv(1024)
    logger.debug('response from server: "%s"', response)

    # Clean up
    logger.debug('closing socket')
    s.close()
    logger.debug('done')
```

The last part is to finalize the server code. The only difference with this version is that the logging has been standardized so that its all python logging and the logging configuration was modified to include dates and write to a file instead of stdout. The listing starts out the same as the other examples:



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

try:
    import logging
    import sys
    import SocketServer
    import MySQLdb
    import threading
    import time
    from daemon import *

except StandardError, e:
    import sys
    print "Error while loading libraries: "
    print e
    sys.exit()

logging.basicConfig(filename='/tmp/dbcheck.log',
                    level=logging.DEBUG,
                    format='%(asctime)s %(name)s: %(message)s',
                    datefmt='%m/%d/%Y %I:%M:%S %p')

```

The request handler is also very similar to what we have seen thus far. At this point, you should probably change the self.MYSQLSVR address to something other than localhost. Localhost will only be reachable via an on server process. To access this remotely you will want this to be one of the interface IP addresses on the server. Credentials, database name, and the query should also be modified to fit the environment.

```python
class EchoRequestHandler(SocketServer.BaseRequestHandler):

    def __init__(self, request, client_address, server):
        self.logger = logging.getLogger('EchoRequestHandler')
        self.logger.debug('__init__')

        # All of the relevant configuration items are kept here
        self.HEALTH_UP = "SERVER_UP"
        self.HEALTH_DOWN = "SERVER_DN"
        self.HEALTH_CHECK = "8675309"

        self.MYSQLSVR = "localhost"
        self.USER = "root"
        self.PWD = "default"
        self.DB = "intel_test"
        self.QUERY = "SELECT * FROM healthcheck"
        self.QUERYTEST = "DB_ALIVE"

        SocketServer.BaseRequestHandler.__init__(self, request, client_address, server)
        return

    def setup(self):
        self.logger.debug('setup')
        return SocketServer.BaseRequestHandler.setup(self)

```

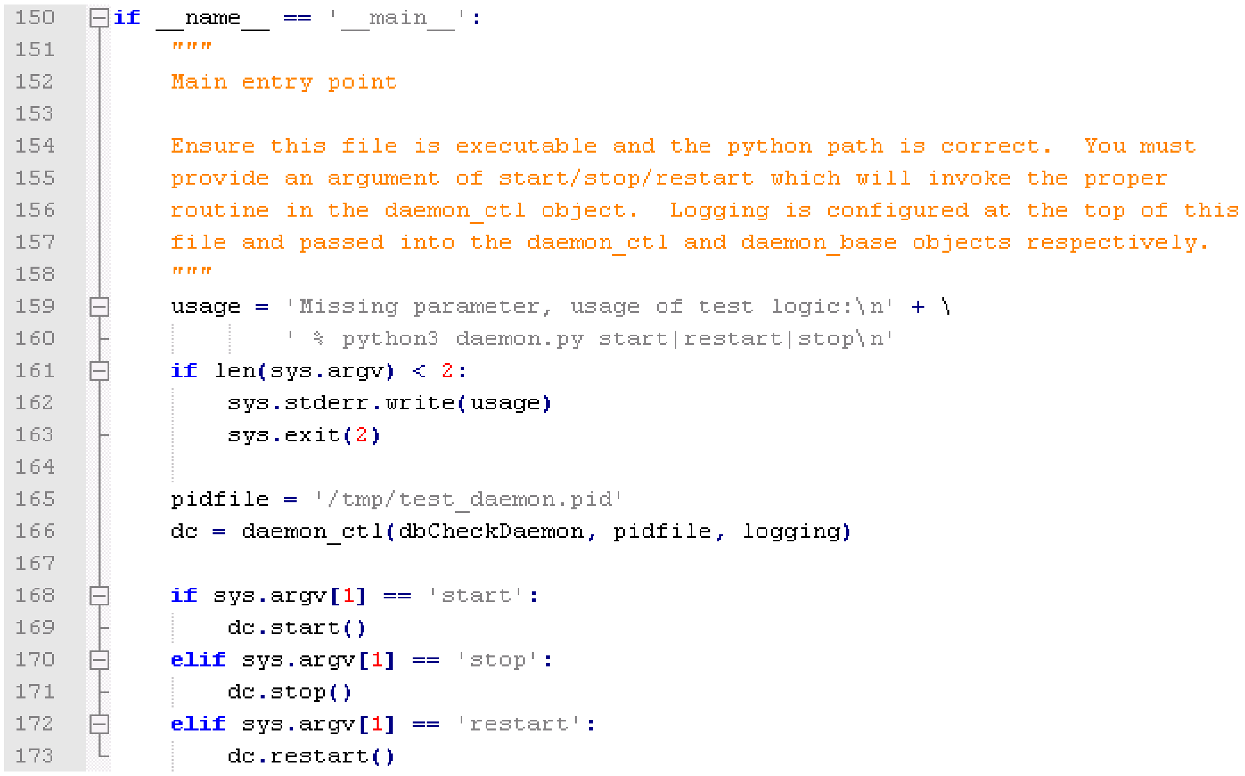
The handle and checkDB functions are the same as previously shown so we won't cover them again. The EchoServer is also created in the same manner as previously:

```python
# -- EchoServer ------------------------------------------------------------ #
# Adds the Threading mixin to implement a threaded TCP Server
# ------------------------------------------------------------------------- #
class EchoServer(SocketServer.ThreadingMixIn, SocketServer.TCPServer):
    pass
```

We create a new daemon class called dbCheckDaemon and override the run method as previously discussed to start our TCP server.

```python
class dbCheckDaemon(daemon_base):
    def run(self):
        self.logger.debug('Starting TCP Server')
        address = ('10.128.1.252', 10888)
        server = EchoServer(address, EchoRequestHandler)
        ip, port = server.server_address

        self.logger.debug('Building main thread')
        t = threading.Thread(target=server.serve_forever)
        t.setDaemon(True)

        t.start()
        self.logger.debug('Server loop running in thread: ', t.getName())

        while True:
            time.sleep(1)

```

Lastly, we update the entry point code so that it reflects the other changes we have made to our code.



```
if __name__ == '__main__':
    """
    Main entry point

    Ensure this file is executable and the python path is correct.  You must
    provide an argument of start/stop/restart which will invoke the proper
    routine in the daemon_ctl object.  Logging is configured at the top of this
    file and passed into the daemon_ctl and daemon_base objects respectively.
    """
    usage = 'Missing parameter, usage of test logic:\n' + \
            ' % python3 daemon.py start|restart|stop\n'
    if len(sys.argv) < 2:
        sys.stderr.write(usage)
        sys.exit(2)

    pidfile = '/tmp/test_daemon.pid'
    dc = daemon_ctl(dbCheckDaemon, pidfile, logging)

    if sys.argv[1] == 'start':
        dc.start()
    elif sys.argv[1] == 'stop':
        dc.stop()
    elif sys.argv[1] == 'restart':
        dc.restart()
```

There are some minor modifications to the daemon_base and daemon_ctl code that can be reviewed independently. The changes are namely for standardizing the logging and providing a means to pass the logging object to the various classes.

Final thoughts

This check is fairly simple but can easily be expanded to perform more. For example, if there are more than one database on the MySQL server you have the option of having a number of these daemons or combining that functionality in one daemon. For example, you could add an additional check that verifies that it's a proper health check and then receives a database name. Depending on that name, it then checks the health of that specific database. You could alternatively health check all the databases and return all the results as a response and then the TCP health check on the F5 would need to sort out what it was looking for.

Other considerations are that the MySQL database check doesn't have to be local to the box, it I possible as long as a there is network access, for the daemon to check a database on a different server. While there are a lot of possibilities it's probably best to maintain as much simplicity as possible.

Lastly, while a lot of effort was made to keep any traffic from the TCP server isolated the database (for security concerns) and exception handling and error checking are liberally used this code is a proof of concept. It does open a port on a server and should you decide to put it into production its advised that you do a thorough code review and ensure that it meets your corporate standards.