Lecture Supplemental: Threading in Python

TCP protocols are often things that can be potentially time-consuming. For example, in Lab 3 we are transferring files; if a file is very large, the transfer could take considerable time.

TCP servers should be able to service multiple clients at a time. But the way the server in Lab 3 was programmed, it can only service one client at a time -- the "`while True`" loop doesn't get to come around and call `accept()` again until the file transfer is complete. Under normal circumstances, this is on acceptable.

So what servers usually do is send the actual client handling code to a new *thread*. A thread is a separate execution stream, allowing a program to (ostensibly) do two or more things at the same time. (Of course, whether these actions are *actually* concurrent, or just scheduled through a rotation of CPU access by the OS, is dependent on the system and not something a programmer has to think about.)

The basic `while True` loop for a server looks like this:

```
while True:
    client, addr = s.accept()
    t = ClientHandler(client)
    t.start()
```

The `ClientHandler()` class is defined as a subclass of `threading.Thread` (that is, the `Thread` class in the module named `threading`). This module is included in Python, but must be imported to be used. `Thread` subclasses define a special object method named `run()`; when the inherited method called `start()` is executed, the `run()` method is then called within a new thread. The `start()` method itself returns *immediately*, so the loop above comes around and calls `accept()` right away, allowing the server to connect to a new client even though the previous client may still be connected and going through it's `run()` method.

The following shows the basic structure of creating a Thread subclass.

```
class ClientHandler(threading.Thread):
    def __init__(self, client):
        threading.Thread.__init__(self)
        self.client = client

    def run(self):
        # here is where the actual client-handling code
        # is placed. Note that you use the variable
        # "self.client" instead of just "client".
        # .
        # .
        # .
```

Python class definitions are a little unusual. Classes were added to Python later, so the structure seems a little hacky. All methods and data members use the "`self`" variable, which acts as a reference to the object itself (just like "`this`" is used in Java, except in Python it is *required*). With methods, you can have other parameters, but `self` has to be the first one.

Python constructors are called "`__init__`", with *two* underscores on other side. Like all methods, "`self`" must be the first parameter, but if your constructor needs more parameters you can add them after the `self`. (For Lab 3, your `ClientHandler` class needs a reference to the client socket, so we pass that as the second argument to the constructor.) The constructor is where you define any persistent data members you need; the parameter is just a local variable and disappears after the constructor is done, so by saving a copy of the reference (e.g. `self.client = client`) you will allow other methods to use it later.

As in Java, a subclass constructor must call a superclass constructor as its first line of code. The difference between Java and Python is that Java is able to call a no-arg superclass constructor *implicitly* (if it exists), so you often don't have to put the superclass constructor call in your subclass constructor. However, in Python it is not optional -- you have to explicitly call the superclass constructor. So that's what's happening with the first line of the `__init__()` method.

Starting Lab 4

The first step is to get the server running the client handling code (everything below the `accept()` method) in its own thread. To do that, make a `ClientHandler` class as shown above, and cut-and-paste all the client handling code into your `run()` method. (Note that the "`client`" variable that you use as a data member must be referenced in your cut-and-paste code as `self.client` instead of just `client`!)

The first line in the `run()` method should be the `send()` call to send "`READY`" to the client. Since you are starting your thread as soon as the connection is established, the `run()` method will be called immediately and the protocol will proceed as usual. Later, though, the client will have to wait until the "manager" decides it's time for the client to run. Note that you can use the same client from Lab 3 to test your Lab 4 server! The client should see the server as no different from the one in Lab 3 -- all the changes are on the efficiency of the server side, and are transparent to the client.

At this point -- before you implement the `Manager` class -- your server loop should look like this:

```
while True:
    client, addr = s.accept()
    t = ClientHandler(client, addr)
    t.start()
```

You create the reference to a `ClientHandler` object, `t`, and then start it. The `start()` method causes the `run()` method of the `ClientHandler` object to execute, but returns immediately, not waiting for the `run()` method to complete. This allows the server socket to `accept()` a new client almost immediately after connecting to the previous one.

Once you have the server able to accept individual clients running in their own threads, you're ready to throw in the `Manager` class. Everything you need to know about the `Manager` is on the Lab 4 specification page. Note that the manager is a separate `threading.Thread` subclass, just like the `ClientHandler`, so all the same setup rules apply: constructor, calling the superclass constructor, and the `run()` method. However, there will only ever be *one* `Manager` thread, created and started at the start of the server code, before the `while True` loop ever begins.